EXHIBIT F

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 7/00, 7/38 — A1

(11) International Publication Number: **WO 97/07450**

(43) International Publication Date: 27 February 1997 (27.02.97)

(21) International Application Number: PCT/US96/13047

(22) International Filing Date: 16 August 1996 (16.08.96)

(30) Priority Data:
08/516,036 16 August 1995 (16.08.95) US

(60) Parent Application or Grant
(63) Related by Continuation
US 08/516,036 (CON)
Filed on 16 August 1995 (16.08.95)

(71) Applicant *(for all designated States except US)*: MICROUNITY SYSTEMS ENGINEERING, INC. [US/US]; 255 Caspian Drive, Sunnyvale, CA 94089-1015 (US).

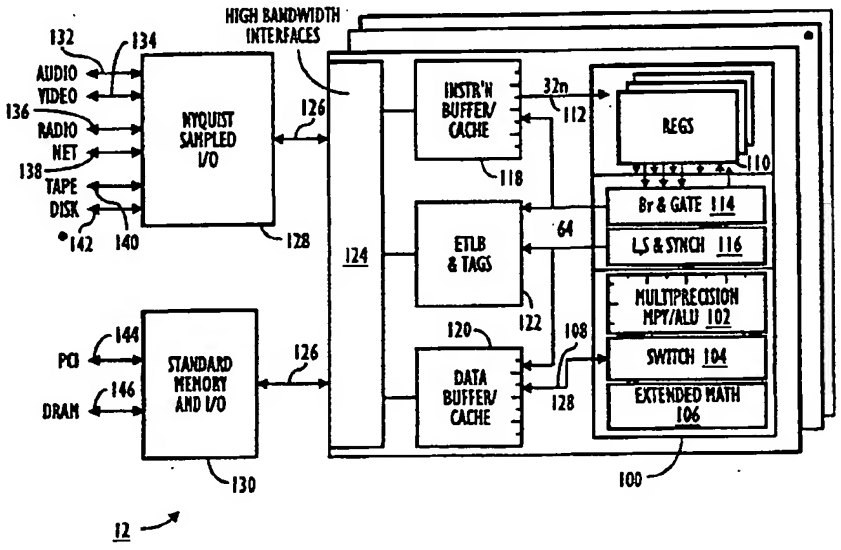
(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: HANSEN, Craig [US/US]; 350 Yerba Santa Avenue, Los Altos, CA 94022 (US). MOUSSOURIS, John [US/US]; 255 Caspian Drive, Sunnyvale, CA 94089 (US).

(74) Agents: FOGARTY, Michael, E. et al.; McDermott, Will and Emery, 1850 K Street, N.W., Washington, DC 20006-2296 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: GENERAL PURPOSE, PROGRAMMABLE MEDIA PROCESSOR



(57) Abstract

A general purpose, programmable media processor (12) for processing and transmitting a media data streams. The media processor (12) incorporates an execution unit (100) that maintains substantially peak data throughout of media data streams. The execution unit (100) includes a dynamically partionable multi-precision arithmetic unit (102), programmable switch (104) and programmable extended mathematical element (106). A high bandwidth external interface (124) supplies media data streams at substantially peak rates to a general purpose register file (110) and the execution unit. A memory management unit, and instruction and data cache/buffers (118, 120). The general purpose, programmable media processor (12) is disposed in a network fabric consisting of fiber optic cable, coaxial cable and twisted pair wires to transmit, process and receive single or unified media data streams.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	GB	United Kingdom	MW	Malawi
AT	Austria	GE	Georgia	MX	Mexico
AU	Australia	GN	Guinea	NE	Niger
BB	Barbados	GR	Greece	NL	Netherlands
BE	Belgium	HU	Hungary	NO	Norway
BF	Burkina Faso	IE	Ireland	NZ	New Zealand
BG	Bulgaria	IT	Italy	PL	Poland
BJ	Benin	JP	Japan	PT	Portugal
BR	Brazil	KE	Kenya	RO	Romania
BY	Belarus	KG	Kyrgystan	RU	Russian Federation
CA	Canada	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic			SE	Sweden
CG	Congo	KR	Republic of Korea	SG	Singapore
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LR	Liberia	SZ	Swaziland
CS	Czechoslovakia	LT	Lithuania	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	LV	Latvia	TJ	Tajikistan
DK	Denmark	MC	Monaco	TT	Trinidad and Tobago
EE	Estonia	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	UG	Uganda
FI	Finland	ML	Mali	US	United States of America
FR	France	MN	Mongolia	UZ	Uzbekistan
GA	Gabon	MR	Mauritania	VN	Viet Nam

WO 97/07450 PCT/US96/13047

GENERAL PURPOSE, PROGRAMMABLE MEDIA PROCESSOR

Field of the Invention

This invention relates to the field of communications processing, and more particularly, to a method and apparatus for real-time processing of multi-media digital communications.

Background of the Invention

Optical fiber and discs have made the transmission and storage of digital information both cheaper and easier than older analog technologies. An improved system for digital processing of media data streams is necessary in order to realize the full potential of these advanced media.

For the past century, telephone service delivered over copper twisted pair has been the lingua franca of communications. Over the next century, broadband services delivered over optical fiber and coax will more completely

fulfill the human need for sensory information by supplying voice, video, and data at rates of about 1,000 times greater than narrow band telephony. Current general-purpose microprocessors and digital signal processors ("DSPs") can handle digital voice, data, and images at narrow band rates, but they are way too slow for processing media data at broadband rates.

This shortfall in digital processing of broadband media is currently being addressed through the design of many different kinds of application-specific integrated circuits ("ASICs"). For example, a prototypical broadband device such as a cable modem modulates and demodulates digital data at rates up to 45 Mbits/sec within a single 6 MHZ cable channel (as compared to rates of 28.8 Kbits/sec within a 6 KHz channel for telephone modems) and transcodes it onto a 10/100baseT connection to a personal computer ("PC") or workstation. Current cable modems thus receive data from a coaxial cable connection through a chain of specialized ASIC devices in order to accomplish Quadrature Amplitude Modification ("QAM") demodulation, Reed-Solomon error correction, packet filtering, Data Encryption Standard ("DES") decryption, and Ethernet protocol handling. The cable modems also transmit data to the coaxial cable link through a second chain of devices to achieve DES encryption, Reed-Solomon block encoding, and Quaternary Phase Shift Keying ("QPSK") modulation. In these environments, a general-purpose processor is usually required as well in order to perform initialization, statistics collection, diagnostics, and network management functions.

The ASIC approach to media processing has three fundamental flaws: cost, complexity, and rigidity. The combined silicon area of all the specialized ASIC devices required in the cable modem, for example, results in a component cost incompatible with the per subscriber price target for a cable service. The cable plant itself is a very hostile service environment, with noise ingress, reflections, nonlinear amplifiers, and other channel impairments, especially when viewed in the upstream direction. Telephony modems have developed an elaborate hierarchy of algorithms implemented in DSP software, with automatic reduction of data rates from 28.8Kbits/sec to 19.6Kbits/sec,

14.4Kbits/sec. or much lower rates as needed to accommodate noise, echoes, and other impairments in the copper plant. To implement similar algorithms on an ASIC-based broadband modem is far more complex to achieve in software.

These problems of cost, complexity, and rigidity are compounded further in more complete broadband devices such as digital set-top boxes, multimedia PCs, or video conferencing equipment, all of which go beyond the basic radio frequency ("RF") modem functions to include a broad range of audio and video compression and decoding algorithms, along with remote control and graphical user interfaces. Software for these devices must control what amounts to a heterogeneous multi-processor, where each specialized processor has a different, and usually eccentric or primitive, programming environment. Even if these programming environments are mastered, the degree of programmability is limited. For example, Motion Picture Expert Group-I ("MPEG-I") chips manufactured by AT&T Corporation will not implement advances such as fractal- and wavelet-based compression algorithms, but these chips are not readily software upgradeable to the MPEG-II standard. A broadband network operator who leases an MPEG ASIC-based product is therefore at risk of having to continuously upgrade his system by purchasing significant amounts of new hardware just to track the evolution of MPEG standards.

The high cost of ASIC-based media processing results from inefficiencies in both memory and logic. A typical ASIC consists of a multiplicity of specialized logic blocks, each with a small memory dedicated to holding the data which comprises the working set for that block. The silicon area of these multiple small memories is further increased by the overhead of multiple decoders, sense amplifiers, write drivers, etc. required for each logic block. The logic blocks are also constrained to operate at frequencies determined by the internal symbol rates of broadband algorithms in order to avoid additional buffer memories. These frequencies typically differ from the optimum speed-area operating point of a given semiconductor technology. Interconnect and synchronization of the many logic and memory blocks are also major sources of overhead in the ASIC approach.

The disadvantages of the prior ASIC approach can be over come by a single unified media processor. The cost advantages of such a unified processor can be achieved by gathering all the many ASIC functions of a broadband media product into a single integrated circuit. Cost reduction is further increased by reducing the total memory area of such a circuit by replacing the multiplicity of small ASIC memories with a single memory hierarchy large enough to accommodate the sum total of all the working sets, and wide enough to supply the aggregate bandwidth needs of all the logic blocks. Additionally, the logic block interconnect circuitry to this memory hierarchy may be streamlined by providing a generally programmable switching fabric. Many of the logic blocks themselves can also replaced with a single multi-precision arithmetic unit, which can be internally partitioned under software control to perform addition, multiplication, division, and other integer and floating point arithmetic operations on symbol streams of varying widths, while sustaining the full data throughput of the memory hierarchy. The residue of logic blocks that perform operations that are neither arithmetic or permutation group oriented can be replaced with an extended math unit that supports additional arithmetic operations such as finite field, ring, and table lookup, while also sustaining the full data throughput of the memory hierarchy.

The above multi-precision arithmetic, permutation switch, and extended math operations can then be organized as machine instructions that transfer their operands to and from a single wide multi-ported register file. These instructions can be further supplemented with load/store instructions that transfer register data to and from a data buffer/cache static random access memory ("SRAM") and main memory dynamic random access memories ("DRAMs"), and with branch instructions that control the flow of instructions executed from an instruction buffer/cache SRAM. Extensions to the load/store instructions can be made for synchronization, and to branch instructions for protected gateways, so that multiple threads of execution for audio, video, radio, encryption, networking, etc. can efficiently and securely share memory and logic resources of a unified machine operating near the optimum speed-area point of the target semiconductor

process. The data path for such a unified media processor can interface to a high speed input/output ("I/O") subsystem that moves media streams across ultra-high bandwidth interfaces to external storage and I/O.

Such a device would incorporate all of the processing capabilities of the specialized multi-ASIC combination into a single, unified processing device. The unified processor would be agile and capable of reprogramming through the transmission of new programs over the communication medium. This programmable, general purpose device is thus less costly than the specialized processor combination, easier to operate and reprogram and can be installed or applied in many differing devices and situations. The device may also be scalable to communications applications that support vast numbers of users through massively parallel distributed computing.

It is therefore an object of this invention to process media data streams by executing operations at very high bandwidth rates.

It is also an object of this invention to unify the audio, video, radio, graphics, encryption, authentication, and networking protocols into a single instruction stream.

It is also an object of this invention to achieve high bandwidth rates in a unified processor that is easy to program and more flexible than a heterogeneous combination of special purpose processors.

It is a further object of the invention to support high level mathematical processing in a unified media processor, including finite group, finite field, finite ring and table look-up operations, all at high bandwidth rates.

It is yet a further object of the invention to provide a unified media processor that can be replicated into a multi-processor system to support a vast array of users.

It is yet another object of this invention to allow for massively parallel systems within the switching fabric to support very large numbers of subscribers and services.

It is also an object of the invention to provide a general purpose programmable processor that could be employed at all points in a network.

It is a further object of this invention to sustain very high bandwidth rates to arbitrarily large memory and input/output systems.

Summary of the Invention

In view of the above, there is provided a system for media processing that maintains substantially peak data throughput in the execution and transmission of multiple media data streams. The system includes in one aspect a general purpose, programmable media processor, and in another aspect includes a method for receiving, processing and transmitting media data streams. The general purpose, programmable media processor of the invention further includes an execution unit, high bandwidth external interface, and can be employed in a parallel multi-processor system.

According to the apparatus of the invention, an execution unit is provided that maintains substantially peak data throughput in the unified execution of multiple media data streams. The execution unit includes a data path, and a multi-precision arithmetic unit coupled to the data path and capable of dynamic partitioning based on the elemental width of data received from the data path. The execution unit also includes a switch coupled to the data path that is programmable to manipulate data received from the data path and provide data streams to the data path. An extended mathematical element is also provided, which is coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput. In a preferred embodiment of the execution unit, at least one register file is coupled to the data path.

According to another aspect of the invention, a general purpose programmable media processor is provided having an instruction path and a data path to digitally process a plurality of media data streams. The media processor includes a high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor. At least one register file is included, which is configurable to receive and store data from the data path and to communicate the stored data to the data

path. A multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate on the data to generate a unified symbol output to the data path.

According to the preferred embodiment of the media processor, means are included for moving data between registers and memory by performing load and store operations, and for coordinating the sharing of data among a plurality of tasks by performing synchronization operations based upon instructions and data received by the execution unit. Means are also provided for securely controlling the sequence of execution by performing branch and gateway operations based upon instructions and data received by the execution unit. A memory management unit operable to retrieve data and instructions for timely and secure communication over the data path and instruction path respectively is also preferably included in the media processor. The preferred embodiment also includes a combined instruction cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time execution of multiple media instruction streams, and a combined data cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time response for multiple media data streams.

In another aspect of the invention, a high bandwidth processor interface for receiving and transmitting a media stream is provided having a data path operable to transmit media information at sustained peak rates. The high bandwidth processor interface includes a plurality of memory controllers coupled in series to communicate stored media information to and from the data path, and a plurality of memory elements coupled in parallel to each of the plurality of memory controllers for storing and retrieving the media information. In the preferred embodiment of the high bandwidth processor interface, the plurality of memory controllers each comprise a paired link disposed between each memory controller, where the paired links each transmit and receive plural bits of data and have differential data inputs and outputs and a differential clock signal.

Yet another aspect of the invention includes a system for unified media processing having a plurality of general purpose media processors, where each media processor is operable at substantially peak data rates and has a dynamically partitioned execution unit and a high bandwidth interface for communicating to memory and input/output elements to supply data to the media processor at substantially peak rates. A bi-directional communication fabric is provided, to which the plurality of media processors are coupled, to transmit and receive at least one media stream comprising presentation, transmission, and storage media information. The bi-directional communication fabric preferably comprises a fiber optic network, and a subset of the plurality of media processors comprise network servers.

According to yet another aspect of the invention, a parallel multi-media processor system is provided having a data path and a high bandwidth external interface coupled to the data path and operable to receive a plurality of data of various sizes from an external source and communicate the received data at a rate that maintains substantially peak operation of the parallel multi-processor system. A plurality of register files, each having at least one register coupled to the data path and operable to store data, are also included. At least one multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate in parallel on data stored in the plurality of register files to generate a unified symbol output for each register file.

According to the method of the invention, unified streams of media data are processed by receiving a stream of unified media data including presentation, transmission and storage information. The unified stream of media data is dynamically partitioned into component fields of at least one bit based on the elemental symbol size of data received. The unified stream of media data is then processed at substantially peak operation.

In one aspect of the invention, the unified stream of media data is processed by storing the stream of unified media data in a general register file.

Multi-precision arithmetic operations can then be performed on the stored stream of unified media data based on programmed instructions, where the multi-precision arithmetic operations include Boolean, integer and floating point mathematical operations. The component fields of unified media data can then be manipulated based on programmed instructions that implement copying, shifting and re-sizing operations. Multi-precision mathematical operations can also be performed on the stored stream of unified media data based on programmed instructions, where the mathematical operations including finite group, finite field, finite ring and table look-up operations. Instruction and data pre-fetching are included to fill instruction and data pipelines, and memory management operations can be performed to retrieve instructions and data from external memory. The instructions and data are preferably stored in instruction and data cache/buffers, in which buffer storage in the instruction and data cache/buffers is dynamically allocated to ensure real-time execution.

Other aspects of the invention include a method for achieving high bandwidth communications between a general purpose media processor and external devices by providing a high bandwidth interface disposed between the media processor and the external devices, in which the high bandwidth interface comprises at least one uni-directional channel pair having an input port and an output port. A plurality of media data streams, comprising component fields of various sizes, are transmitted and received between the media processor and the external devices at a rate that sustains substantially peak data throughput at the media processor. A method for processing streams of media data is also included that provides a bi-directional communications fabric for transmitting and receiving at least one stream of media data, where the at least one stream of media data comprises presentation, transmission and storage information. At least one programmable media processor is provided within the communications network for receiving, processing and transmitting the at least one stream of unified media data over the bi-directional communications fabric.

The general purpose, programmable media processor of the invention combines in a single device all of the necessary hardware included in the

specialized processor combinations to process and communicate digital media data streams in real-time. The general purpose, programmable media processor is therefore cheaper and more flexible than the prior approach to media processing. The general purpose, programmable media processor is thus more susceptible to incorporation within a massively parallel processing network of general purpose media processors that enhance the ability to provide real-time multi-media communications to the masses.

These features are accomplished by deploying server media processors and client media processors throughout the network. Such a network provides a seamless, global media super-computer which allows programmers and network owners to virtualize resources. Rather than restrictively accessing only the memory space and processing time of a local resource, the system allows access to resources throughout the network. In small access points such as wireless devices, where very little memory and processing logic is available due to limited battery life, the system is able to draw upon the resources of a homogeneous multi-computer system.

The invention also allows network owners the facility to track standards and to deploy new services by broadcasting software across the network rather than by instituting costly hardware upgrades across the whole network. Broadcasting software across the network can be performed at the end of an advertisement or other program that is broadcasted nationally. Thus, services can be advertised and then transmitted to new subscribers at the end of the advertisement.

These and other features and advantages of the invention will be apparent upon consideration of the following detailed description of the presently preferred embodiments of the invention, taken in conjunction with the appended drawings.

Brief Description of the Drawings

FIG. 1 is a block diagram of a broad band media computer employing the general purpose, programmable media processor of the invention;

FIG. 2 is a block diagram of a global media processor employing multiple general purpose media processors according to the invention;

FIG. 3 is an illustration of the digital bandwidth spectrum for telecommunications, media and computing communications;

FIG. 4 is the digital bandwidth spectrum shown in FIG. 3 taking into account the bandwidth overhead associated with compressed video techniques;

FIG. 5 is a block diagram of the current specialized processor solution for mass media communication, where FIG. 5(a) shows the current distributed system, and FIG. 5(b) shows a possible integrated approach;

FIG. 6 is a block diagram of two presently preferred general purpose media processors, where FIG. 6(a) shows a distributed system and FIG. 6(b) shows an integrated media processor;

FIG. 7 is a block diagram of the presently preferred structure of a general purpose, programmable media processor according to the invention;

FIG. 8 is a drawing consisting of visual illustrations of the various group operations provided on the media processor, where FIG. 8(a) illustrates the group expand operation, FIG. 8(b) illustrates the group compress or extract operation, FIG. 8(c) illustrates the group deal and shuffle operations, FIG. 8(d) illustrates the group swizzle operation and FIG. 8(e) illustrates the various group permute operations;

FIG. 9 shows the preferred instruction and data sizes for the general purpose, programmable media processor, where FIG. 9(a) is an illustration of the various instruction formats available on the general purpose, programmable media processor, FIG. 9(b) illustrates the various floating-point data sizes available on the general purpose media processor, and FIG. 9(c) illustrates the various fixed-point data sizes available on the general purpose media processor;

FIG. 10 is an illustration of a presently preferred memory management unit included in the general purpose processor shown in FIG. 7, where FIG. 10(a) is a translation block diagram and FIG. 10(b) illustrates the functional blocks of the transaction lookaside buffer;

FIG. 11 is an illustration of a super-string pipeline technique;

FIG. 12 is an illustration of the presently preferred super-spring pipeline technique;

FIG. 13 is a block diagram of a single memory channel for communication to the general purpose media processor shown in FIG. 7;

FIG. 14 is an illustration of the presently preferred connection of standard memory devices to the preferred memory interface;

FIG. 15 is a block diagram of the input/output controller for use with the memory channel shown in FIG. 13;

FIG. 16 is a block diagram showing multiple memory channels connected to the general purpose media processor shown in FIG. 7, where FIG. 16(a) shows a two-channel implementation and FIG. 16(b) illustrates a twelve-channel embodiment;

FIG. 17 illustrates the presently preferred packet communications protocol for use over the memory channel shown in FIG. 13;

FIG. 18 shows a multi-processor configuration employing the general purpose media processor shown in FIG. 7, where FIG. 18(a) shows a linear processor configuration, FIG. 18(b) shows a processor ring configuration, and FIG. 18(c) shows a two-dimensional processor configuration; and

FIG. 19 shows a presently preferred multi-chip implementation of the general purpose, programmable media processor of the invention.

Detailed Description of the Presently Preferred Embodiments

Referring to the drawings, where like-reference numerals refer to like elements throughout, a broad band microcomputer 10 is provided in FIG. 1. The broad band microcomputer 10 consists essentially of a general purpose media processor 12. As will be described in more detail below, the general purpose media processor 12 receives, processes and transmits media data streams in a bi-directional manner from upstream network components to downstream devices. In general, media data streams received from upstream network components can comprise any combination of audio, video, radio, graphics, encryption, authentication, and networking information. As those skilled in the art will

appreciate, however, the general purpose media processor 12 is in no way limited to receiving, processing and transmitting only these types of media information. The general purpose media processor 12 of the invention is capable of processing any form of digital media information without departing from the spirit and essential scope of the invention.

System Configuration

In the preferred embodiment of the invention shown in FIG. 1, media data streams are communicated to the media processor 12 from several sources. Ideally, unified media data streams are received and transmitted by the general purpose media processor 12 over a fiber optic cable network 14. As will be described in more detail below, although a fiber optic cable network is preferred, the presently existing communications network in the United States consists of a combination of fiber optic cable, coaxial cable and other transmission media. Consequently, the general purpose media processor 12 can also receive and transmit media data streams over coaxial cable 14 and traditional twisted pair wire connections 16. The specific communications protocol employed over the twisted pair 16, whether POTS, ISDN or ADSL, is not essential; all protocols are supported by the broad band microcomputer 10. The details of these protocols are generally known to those skilled in the art and no further discussion is therefore needed or provided herein.

Another form of upstream network communication is through a satellite link 18. The satellite link 18 is typically connected to a satellite receiver 20. The satellite receiver 20 comprises an antenna, usually in the form of a satellite dish, and amplification circuitry. The details of such satellite communications are also generally known in the art, and further detail is therefore not provided or included herein.

As described above, the general purpose media processor 12 communicates in a bi-directional manner to receive, process and transmit media data streams to and from downstream devices. As shown in FIG. 1, downstream communication preferably takes place in at least two forms. First, media data

streams can be communicated over a bi-directional local network 22. Various types of local networks 22 are generally known in the art and many different forms exist. The general purpose media processor 12 is capable of communicating over any of these local networks 22 and the particular type of network selected is implementation specific.

The local network 22 is preferably employed to communicate between the unified processor 12, and audio/visual devices 24 or other digital devices 26. Presently preferred examples of audio/visual devices 24 include digital cable television, video-on-demand devices, electronic yellow pages services, integrated message systems, video telephones, video games and electronic program guides. As those skilled in the art will appreciate, other forms of audio/video devices are contemplated within the spirit and scope of the invention. Presently preferred embodiments of other digital devices 26 for communication with the general purpose media processor 12 include personal computers, television sets, work stations, digital video camera recorders, and compact disc read-only memories. As those skilled in the art will also appreciate, further digital devices 26 are contemplated for communication to the general purpose media processor 12 without departing from the spirit and scope of the invention.

Second, the general purpose media processor preferably also communicates with downstream devices over a wireless network 28. In the presently preferred embodiment of the invention, wireless devices for communication over the wireless network 28 can comprise either remote communication devices 30 or remote computing devices 32. Presently preferred embodiments of the remote communications devices 30 include cordless telephones and personal communicators. Presently preferred embodiments of the remote computing devices 32 include remote controls and telecommunicating devices. As those skilled in the art will appreciate, other forms of remote communication devices 30 and remote computing devices 32 are capable of communication with the general purpose media processor 12 without departing from the spirit and scope of the invention. An agile digital radio (not shown) that incorporates a

general purpose media processor 12 may be used to communicate with these wireless devices.

Network Configuration

Referring now to FIG. 2, the general purpose media processor 12 is preferably disposed throughout a digital communications network 38. In order to enable communication among large and small businesses, residential customers and mobile users, the network 38 can consist of a combination of many individual sub-networks comprised of three main forms of interconnection. The trunk and main branches of the network 38 preferably employ fiber optic cable 40 as the preferred means of interconnection. Fiber optic cable 40 is used to connect between general purpose media processors 12 disposed as network servers 46 or large business installations 48 that are capable of coupling directly to the fiber optic link 40. For communications to small business and residential customers that may be incapable of directly coupling to the fiber optic cable 40, a general purpose media processor 12 can be used as an interface to other forms of network interconnection.

As shown in FIG. 2, alternate forms of interconnection consist of coaxial cable lines 42 and twisted pair wiring 44. Coaxial cable lines are currently in place throughout the U.S. and is typically employed to provide cable television services to residential homes. According to the preferred embodiment of the invention, general purpose media processors 12 can be installed at these residential locations 52. In contrast to the specialized processor approach, the general purpose media processor 12 provides enough bandwidth to allow for bi-directional communications to and from these residential locations 52.

Network servers 46 controlled by general purpose media processors 12 are also employed throughout the network 38. For example, the network servers 46 can be used to interface between the fiber optic network 40 and twisted pair wiring 44. Twisted pair wiring 44 is still employed for small businesses 50 and residential locations 52 that do not or cannot currently subscribe to coaxial cable or fiber optic network services. General purpose media processors 12 are also disposed at these small business locations 50 and non-cable residential

locations 52. General purpose media processors 12 are also installed in wireless or mobile locations 52, which are coupled to the network 38 through agile digital radios (not shown). As shown in FIG. 2, network databases or other peripherals 56 can also coupled to general purpose media processors 12 in the network 38.

The general purpose media processor 12 is operable at significantly high bandwidths in order to receive, process and transmit unified media data streams. Referring to FIG. 3, the respective frequencies for various types of media data streams are set forth against a bandwidth spectrum 60. The bandwidth spectrum 60 includes three component spectrums, all along the same range of frequencies, which represent the various frequency rates of digital media communications. Current computing bandwidth capabilities are also displayed. The telecommunications spectrum 62 shows the various frequency bands used for telecommunications transmission. For example, teletype terminals and modems operate in a range between approximately 64 bits/second to 16 kilobits/second. The ISDN telecommunication protocol operates at 64 kilobits/second. At the upper end of the telecommunications spectrum 62, T1 and T3 trunks operate at one megabit per second and 32 megabits per second, respectively. The SONET frequency range extends from approximately 128 megabits per second up to approximately 32 gigabits per second. Accordingly, in order to carry such broad band communications, the general purpose media processor 12 is capable of transferring information at rates into the gigabits per second range or higher.

A spectrum of typical media data streams is presented in the media spectrum 64 shown in FIG. 3. Voice and music transmissions are centered at frequencies of approximately 64 kilobits per second and one megabit per second, respectively. At the upper end of the media spectrum 64, video transmission takes place in a range from 128 megabits per second for high density television up to over 256 gigabits per second for movie applications. When using common video compression techniques, however, the video transmission spectrum can be shifted down to between 32 kilobits per second to 128 megabits per second as a result of the data compression. As described below, the processing required to achieve the data compression results in an increase in bandwidth requirements.

Current computing bandwidths are shown in the computing spectrum 66 of FIG. 3. Serial communications presently take place in a range between two kilobits per second up to 512 kilobits per second. The Ethernet network protocol operates at approximately 8 megabits per second. Current dynamic random access memory and other digital input/output peripherals operate between 32 megabits per second and 512 megabits per second. Presently available microprocessors are capable of operation in the low gigabits per second range. For example, the '386 Pentium microprocessor manufactured by Intel Corporation of Santa Clara, California operates in the lower half of that range, and the Alpha microprocessor manufactured by Digital Equipment Corporation approaches the 16 gigabits per second range.

When video compression is employed, as expressed above, the associated processing overhead reduces the effective bandwidth of the particular processor. As a result, in order to handle compressed video, these processors must operate in the terahertz frequency range. The bandwidth spectrum 60 shown in FIG. 4 represents the effect of handling media data streams including compressed video. The computing spectrum 66 is skewed down to properly align the computing bandwidth requirements with the telecommunications spectrum 62 and the media spectrum 64. Accordingly, current processor technology is not sufficient to handle the transmission and processing associated with complex streams of multi-media data.

The current specialized processor approach to media processing is illustrated in the block diagram shown in FIG. 5. As shown in FIG. 5, special purpose processors are coupled to a back plane 70, which is capable of transmitting instructions and data at the upper kilobits to lower gigabits per second range. In a typical configuration, an audio processor 76, video processor 78, graphics processor 80 and network processor 82 are all coupled to the back plane 70. Each of the audio, video, graphics and network processors 76-82 typically employ their own private or dedicated memories 84, which are only accessible to the specific processor and not accessible over the back plane 70. As described above, however, unless video data streams are constantly being processed, for

example, the video processor 78 will sit idle for periods of time. The computing power of the dedicated video processor 78 is thus only available to handle video data streams and is not available to handle other media data streams that are directed to other dedicated processors. This, of course, is an inefficient use of the video processor 78 particularly in view of the overall processing capability of this multi-processor system.

The general purpose media processor 12, in contrast, handles a data stream of audio, video, graphics and network information all at the same time with the same processor. In order to handle the ever changing combination of data types, the general purpose media processor 12 is dynamically partitionable to allocate the appropriate amount of processing for each combination of media in a unified media data stream. A block diagram of two preferred general purpose media processor system configurations is shown in FIG. 6. Referring to FIG. 6(a), a general purpose media processor 12 is coupled to a high-speed back plane 90. The presently preferred back plane 90 is capable of operation at 30 gigabits per second. As those skilled in the art will appreciate, back planes 90 that are capable of operation at 400 gigabits per second or greater bandwidth are envisioned within the spirit and scope of the invention. Multiple memory devices 92 are also coupled to the back plane 90, which are accessible by the general purpose media processor 12. Input/output devices 94 are coupled to the back plane 90 through a dual-ported memory 92. The configuration of the input/output devices 94 on one end of the dual-ported memory 92 allows the sharing of these memory devices 92 throughout a network 38 of general purpose media processors 12.

Alternatively, FIG. 6(b) shows a presently preferred integrated general purpose media processor 12. The integrated processor includes on-board memory and I/O 86. The on-board memory is preferably of sufficient size to optimize throughput, and can comprise a cache and/or buffer memory or the like. The integrated media processor 12 also connects to external memory 88, which is preferably larger than the on-board memory 86 and forms the system main memory.

Execution Unit

One presently preferred embodiment of an integrated general purpose media processor 12 is shown in FIG. 7. The core of the integrated general purpose media processor 12 comprises an execution unit 100. Three main elements or subsections are included in the execution unit 100. A multiple precision arithmetic/logic unit ("ALU") 102 performs all logical and simple arithmetic operations on incoming media data streams. Such operations consist of calculate and control operations such as Boolean functions, as well as addition, subtraction, multiplication and division. These operations are performed on single or unified media data streams transmitted to and from the multiple precision ALU 102 over a data bus or data path 108. Preferably the data path 108 is 128 bits wide, although those skilled in the art will appreciate that the data path 108 can take on any width or size without departing from the spirit and scope of the invention. The wider the data path 108 the more unified media data can be processed in parallel by the general purpose media processor 12.

Coupled to the multi-precision ALU 102 via the data path 108, and also an element of the execution unit 100, is a programmable switch 104. The programmable switch 104 performs data handling operations on single or unified media data streams transmitted over the data path 108. Examples of such data handling operations include deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, although other data handling operations are contemplated. These operations can be performed on single bits or bit fields consisting of two or more bits up to the entire width of the data path 108. Thus, single bits or bit fields of various sizes can be manipulated through programmable operation of the switch 104.

Examples of the presently preferred data manipulation operations performed by the general purpose media processor 12 are shown in FIG. 8. A group expand operation is visually illustrated in FIG. 8(a). According to the group expand operation, a sequential field of bits 270 can be divided into constituent sub-fields 272a-272d for insertion into a larger field array 274. The reverse of the group expand operation is a group compress or extract operation. A

visual illustration of the group compress or extract operation is shown in FIG. 8(b). As shown, separate sub-fields 272a-272d from a larger bit field 274 can be combined to form a contiguous or sequential field of bits 270.

Referring to FIGS. 8(c)-8(e), group deal, shuffle, swizzle and permute operations performed by the programmable switch 104 are also illustrated. The operations performed by these instructions are readily understood from a review of the drawings. The group manipulation operations illustrated in FIGS. 8(a)-8(e) comprise the presently contemplated data manipulation operations for the general purpose media processor 12. As those skilled in the art will appreciate, either a subset of these operations or additional data manipulation operations can be incorporated in other alternate embodiments of the general purpose media processor 12 without departing from the spirit and scope of the invention.

Referring again to FIG. 7, higher level mathematical operations than those performed by the multi-precision ALU 102 are performed in the general purpose media processor 12 through an extended math element 106. The extended math element 106 is coupled to the data path 108 and also comprises part of the execution unit 100. The extended math element 106 performs the complex arithmetic operations necessary for video data compression and similarly intensive mathematical operations. One presently preferred example of an extended math operation comprises a Galois field operation. Other examples of extended mathematical functions performed by the extended math element 106 include CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding. As those skilled in the art appreciate, additional mathematical operations are possible and contemplated.

According to the preferred embodiment of the integrated general purpose media processor 12, a register file 110 is provided in addition to the execution unit 100 to process media data. The register file 110 stores and transmits data streams to and from the execution unit 100 via the data path 108. Rather than employing a complex set of specific or dedicated registers, the general purpose media processor 12 preferably includes 64 general purpose registers in the

register file 110 along with one program counter (not shown). The 64 general purpose registers contained in the register file 110 are all available to the user/programmer, and comprise a portion of the user state of the general purpose media processor 12. The general purpose registers are preferably capable of storing any form of data. Each register within the register file 110 is coupled to the data path 108 and is accessible to the execution unit 100 in the same manner. Thus, the user can employ a general purpose register according to the specific needs of a particular program or unique application. As those skilled in the art will appreciate, the register file 110 can also comprise a plurality of register files 110 configured in parallel in order to support parallel multi-threaded processing.

Instruction Set and User Programming

Control or manipulation of data processed by the general purpose media processor 12 is achieved by selected instructions programmed by the user. Those skilled in the art will appreciate that a great number of programs are possible through various sequences of instructions. Particular programs can be developed for each unique implementation of the general purpose media processor 12. A detailed discussion of such specific programs is therefore beyond the scope of this description.

One presently preferred instruction set for the general purpose media processor 12 is included in the Microfiche Appendix, the contents of which are hereby incorporated herein by reference. A list of the presently preferred major operation codes for the general purpose media processor 12 appears below in Table I.

MAJOR OPERATION CODES

MAJOR	0	32	64	96	128	160	192	224
0	ERES	GSHUFFLEI	FMULADD16I	GMULADD1	LU16LAI	SAAS64LAI	EADDIO	BFE16
1	ESHUFFLE[illegible]MUX	GSHUFFLE[illegible]MUX	FMULADD32I	GMULADD2	LU16BAI	SAAS64BAI	EADDIUO	BFNUE16
2		GSELECT8	FMULADD64I	GMULADD4	LU16LI	SCAS64LAI	ESETIL	BFNUGE16
3	EMDEPI	GMDEPI		GMULADD8	LU16BI	SCAS64BAI	ESETIGE	BFNUL16
4	EMUX	GMUX	FMULSUB16I	GMULADD16I	LU32LAI	SMAS64LAI	ESETIE	BFE32
5	E8MUX	G8MUX	FMULSUB32I	GMULADD32I	LU32BAI	SMAS64BAI	ESETINE	BFNUE32
6	EGFMUL64	GGFMUL8	FMULSUB64I	GMULADD64I	LU32LI	SMUX64LAI	ESETIUL	BFNUGE32
7	ETRANSPOSEMUX	GTRANSPOSEMUX		GEXTRACT128I	LU32BI	SMUX64BAI	ESETIUGE	BFNUL32
8					L16LAI	S16LAI	ESUBIO	BFE64
9	ESWIZZLE	GSWIZZLE		GUMULADD2	L16BAI	S16BAI	ESUBIUO	BFNUE64
10		GSWIZZLECOPY		GUMULADD4	L16LI	S16LI	ESUBIL	BFNUGE64
11		GSWIZZLESWAP		GUMULADD8	L16BI	S16BI	ESUBIGE	BFNUL64
12	EDEPI	GDEPI	F.16	GUMULADD16I	L32LAI	S32LAI	ESUBIE	BFE128
13	EUDEPI	GUDEPI	F.32	GUMULADD32I	L32BAI	S32BAI	ESUBINE	BFNUE128
14	EWTHI	GWTHI	F.64	GUMULADD64I	L32LI	S32LI	ESUBIUL	BFNUGE128
15	EUWTHI	GUWTHI		GUEXTRACT128I	L32BI	S32BI	ESUBIUGE	BFNUL128
16			GFMULADD16I	GEXTRACTI	L64LAI	S64LAI	EADDI	BANDE
17			GFMULADD32I	GEXTRACTI16	L64BAI	S64BAI	EXORI	BANDNE
18			GFMULADD64I	GEXTRACTI32	L64LI	S64LI	EORI	BL/BLZ
19			GFMULADD128I	GEXTRACTI64	L64BI	S64BI	EANDI	BGE/BGEZ
20			GFMULSUB16I	GEXTRACT	L128LAI	S128LAI	ESUBI	BE
21			GFMULSUB32I	I32	L128BAI	S128BAI		BNE
22			GFMULSUB64I	G.EXTRACT	L128LI	S128LI	ENORI	BUL/BGZ
23			GFMULSUB128I	I128	L128BI	S128BI	ENANDI	BUGE/BLEZ
24				G I	L8I	S8I		BGATEI
25				G 2	LU8I			
26				G 4				
27				G 8				
28		ECOPYI	GF.16	G.16			ECOPYI	BI
29			GF.32	G 32				BLINKI
30			GF.64	G.64				
31		E.MINOR	GF.128	G.128	L.MINOR	S.MINOR	E MINOR	B MINOR

major operation code field values

TABLE I

As shown in Table I, the major operation codes are grouped according to the function performed by the operations. The operations are thus arranged and listed above according to the presently preferred operation code number for each instruction. As many as 255 separate operations are contemplated for the preferred embodiment of the general purpose media processor 12. As shown in Table I, however, not all of the operation codes are presently implemented. As those skilled in the art will appreciate, alternate schemes for organizing the operation codes, as well as additional operation codes for the general purpose media processor 12, are possible.

The instructions provided in the instruction set for the general purpose media processor 12 control the transfer, processing and manipulation of data streams between the register file 110 and the execution unit 100. The presently preferred width of the instruction path 112 is 32-bits wide, organized as four eight-bit bytes ("quadlets"). Those skilled in the art will appreciate, however, that the instruction path 112 can take on any width without departing from the spirit and scope of the invention. Preferably, each instruction within the instruction set is stored or organized in memory on four-byte boundaries. The presently preferred format for instructions is shown in FIG. 9(a).

As shown in FIG. 9(a), each of the presently preferred instruction formats for the general purpose media processor 12 includes a field 280 for the major operation code number shown in Table I. Based on the type of operation performed, the remaining bits can provide additional operands according to the type of addressing employed with the operation. For example, the remainder of the 32-bit instruction field can comprise an immediate operand ("imm"), or operands stored in any of the general registers ("ra," "rb," "rc," and "rd"). In addition, minor operation codes 282 can also be included among the operands of certain 32-bit instruction formats.

The presently preferred embodiment of the general purpose media processor 12 includes a limited instruction set similar to those seen in Reduced Instruction Set Computer ("RISC") systems. The preferred instruction set for the general purpose media processor 12 shown in Table I includes operations which implement load, store, synchronize, branch and gateway functions. These five groups of operations can be visually represented as two general classes of related operations. The branch and gateway operations perform related functions on media data streams and are thus visually represented as block 114 in FIG. 7. Similarly, the load, store and synchronize operations are grouped together in block 116 and perform similar operations on the media data streams. (Blocks 114 and 116 only represent the above classification of these operations and their function in the processing of media data streams, and do not indicate any specific underlying electronic connections.) A more detailed discussion of these operations, and the

functionality of the general purpose media processor 12, appears in the Microfiche Appendix.

The four-byte structure of instructions for the general purpose media processor 12 is preferably independent of the byte ordering used for any data structures. Nevertheless, the gateway instructions are specifically defined as 16-byte structures containing a code address used to securely invoke a procedure at a higher privilege level. Gateways are preferably marked by protection information specified in the translation lookaside buffer 148 in the memory management unit 122. Gateways are thus preferably aligned on 16-byte boundaries in the external memory. In addition to the general purpose registers and program counter, a privilege level register is provided within the register file 110 that contains the privilege level of the currently executing instruction.

The instruction set preferably includes load and store instructions that move data between memory and the register file 110, branch instructions to compare the content of registers and transfer control, and arithmetic operations to perform computations on the contents of registers. Swap instructions provide multi-thread and multi-processor synchronization. These operations are preferably indivisible and include such instructions as add-and-swap, compare-and-swap, and multiplex-and-swap instructions. The fixed-point compare-and-branch instructions within the instruction set shown in Table I provide the necessary arithmetic tests for equality and inequality of signed and unsigned fixed-point values. The branch through gateway instruction provides a secure means to access code at a higher privileged level in a form similar to a high level language procedure call generally known in the art.

The general purpose media processor 12 also preferably supports floating-point compare-and-branch instructions. The arithmetic operations, which are supported in hardware, include floating-point addition, subtraction, multiplication, division and square root. The general purpose media processor 12 preferably supports other floating-point operations defined by the ANSI-IEEE floating-point standard through the use of software libraries. A floating point

value can preferably be 16, 32, 64 or 128-bits wide. Examples of the presenting preferred floating-point data sizes are illustrated in FIG. 9(b).

The general purpose media processor 12 preferably supports virtual memory addressing and virtual machine operation through a memory management unit 122. Referring to FIG. 10(a), one presently preferred embodiment of the memory management unit 122 is shown. The memory management unit 122 preferably translates global virtual addresses into physical addresses by software programmable routines augmented by a hardware translation lookaside buffer ("TLB") 148. A facility for local virtual address translation 164 is also preferably provided. As those skilled in the art will appreciate, the memory management unit 122 includes a data cache 166 and a tag cache 168 that store data and tags associated with memory sections for each entry in the TLB 148.

A block diagram of one preferred embodiment of the TLB 148 is shown in FIG. 10(b). The TLB 148 receives a virtual address 230 as its input. For each entry in the TLB 148, the virtual address 230 is logically AND-ed with a mask 232. The output of each respective AND gate 234 is compared via a comparator 236 with each entry in the TLB 148. If a match is detected, an output from the comparator 236 is used to gate data 240 through a transceiver 238. As those skilled in the art will appreciate, a match indicates the entry of the corresponding physical address within the contents of the TLB 148 and no external memory or I/O access is required. The data 240 for the data cache 166 (FIG. 10(a)) is then combined with the remaining lower bits of the virtual address 230 through an exclusive-OR gate 242. The resultant combination is the physical address 244 output from the TLB 148. If a match is not detected between the logical address and the contents of the tag cache 168, the memory management unit 122 an external memory or I/O access is necessary to retrieve the relevant portion of memory and update the contents of the TLB 148 accordingly.

Using generally known memory management techniques, the memory management unit 122 ensures that instructions (and data) are properly retrieved from external memory (or other sources) over an external input/output bus 126 (see FIG. 7). As described in more detail below, a high bandwidth

interface 124 is coupled to the external input/output bus 126 to communicate instructions (and media data streams) to the general purpose media processor 12. The presently preferred physical address width for the general purpose media processor 12 is eight bytes (64-bits). In addition, the memory management unit 122 preferably provides match bits (not shown) that allow large memory regions to be assigned a single TLB entry allowing for fine grain memory management of large memory sections. The memory management unit 122 also preferably includes a priority bit (not shown) that allows for preferential queuing of memory areas according to respective levels of priority. Other memory management operations generally known in the art are also performed by the memory management unit 122.

Referring again to FIG. 7, instructions received by the general purpose media processor 12 are stored in a combined instruction buffer/cache 118. The instruction buffer/cache 118 is dynamically subdivided to store the largest sequence of instructions capable of execution by the execution unit 100 without the necessity of accessing external memory. In a preferred embodiment of the invention, instruction buffer space is allocated to the smallest and most frequently executed blocks of media instructions. The instruction buffer thus helps maintain the high bandwidth capacity of the general purpose media processor 12 by sustaining the number of instructions executed per second at or near peak operation. That portion of the instruction buffer/cache 118 not used as a buffer is, therefore, available to be used as cache memory. The instruction buffer/cache 118 is coupled to the instruction path 112 and is preferably 32 kilobytes in size.

A data buffer/cache 120 is also provided to store data transmitted and received to and from the execution unit 100 and register file 110. The data buffer/cache 120 is also dynamically subdivided in a manner similar to that of the instruction buffer/cache 118. The buffer portion of the data buffer/cache 120 is optimized to store a set size of unified media data capable of execution without the necessity of accessing external memory. In a preferred embodiment of the invention, data buffer space is allocated to the smallest and most frequently accessed working sets of media data. Like the instruction buffer, the data buffer

thus maintains peak bandwidth of the general purpose media processor 12. The data buffer/cache 120 is coupled to the data path 108 and is preferably also 32 kilobytes in size.

The preferred embodiment of the general purpose media processor 12 includes a pipelined instruction pre-fetch structure. Although pipelined operation is supported, the general purpose media processor 12 also allows for non-pipelined operations to execute without any operational penalty. One preferred pipeline structure for the general purpose media processor 12 comprises a "super-string" pipeline shown in FIG. 11. A super-string pipeline is designed to fetch and execute several instructions in each clock cycle. The instructions available for the general purpose media processor 12 can be broken down into five basic steps of operation. These steps include a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store and a branch operation. According to the super-string pipeline organization of the general purpose media processor 12, one instruction from each of these five types may be issued in each clock cycle. The presently preferred ordering of these operations are as listed above where each of the five steps are assigned letters "A," "L," "E," "S" and "B" (see FIG. 11).

According to the super-string pipelining technique, each of the instructions are serially dependent, as shown in FIG. 11, and the general purpose media processor 12 has the ability to issue a string of dependent instructions in a single clock cycle. These instructions shown in FIG. 11 can take from two to five cycles of latency to execute, and a branch prediction mechanism is preferably used to keep up the pipeline filled (described below). Instructions can be encoded in unit categories such as address, load, store/sync, fixed, float and branch to allow for easy decoding. A similar scheme is employed to pre-fetch data for the general purpose media processor 12.

As those skilled in the art will appreciate, the super-string pipeline can be implemented in a multi-threaded environment. In such an implementation, the number of threads is preferably relatively prime with respect to functional unit

rates so that functional units can be scheduled in a non-interfering fashion between each thread.

In another more preferred embodiment, a "super-spring" pipelining scheme is employed with the general purpose media processor 12. The super-spring pipeline technique breaks the super-string pipeline shown in FIG. 11 into two sections that are coupled via a memory buffer (not shown). A visual representation of the super-spring pipeline technique is shown in FIG. 12. The front of the pipeline 204, in which address calculation (A), memory load (L), and branch (B) operations are handled, is decoupled from the back of the pipeline 206, in which data calculation (E) and memory store (S) operations are handled. The decoupling is accomplished through the memory buffer (not shown), which is preferably organized in a first-in-first-out ("FIFO") fast/dense structure. (The memory buffer is functionally represented as a spring in FIG. 12.)

As indicated in Table I above, the general purpose media processor 12 does not include delayed branch instructions, and so relies upon branch or fetch prediction techniques to keep the pipeline full in program flows around unconditional and conditional branch instructions. Many such techniques are generally known in the art. Examples of some presently preferred techniques include the use of group compare and set, and multiplex operations to eliminate unpredictable branches; the use of short forward branches, which cause pipeline neutralization; and where branch and link predicts the return address in a one or more entry stack. In addition, the specialized gateway instructions included in the general purpose media processor 12 allow for branches to and from protected virtual memory space. The gateway instructions, therefore, allow an efficient means to transfer between various levels of privilege.

As described above, two basic forms of media data are processed by the general purpose media processor 12, as shown in FIG. 7. These data streams generally comprise Nyquist sampled I/O 128, and standard memory and I/O 130. As shown in FIG. 7, audio 132, video 134, radio 136, network 138, tape 140 and disc 142 data streams comprise some examples of digitally sampled I/O 128. As those skilled in the art will appreciate, other forms of digitally sampled I/O are

contemplated for processing by the general purpose media processor 12 without departing from the spirit and scope of the invention. Standard memory and I/O 130 comprises data received and transmitted to and from general digital peripheral devices used in the design of most computer systems. As shown in FIG. 7, some examples of such devices include dynamic random access memory ("DRAM") 146, or any data received over the PCI bus 144 generally known in the art. Other forms of standard memory and I/O sources are also contemplated. The various fixed-point data sizes preferred for the general purpose media processor 12 are illustrated in FIG. 9(c).

External Interface

As mentioned above, the general purpose media processor 12 includes a high bandwidth interface 124 to communicate with external memory and input/output sources. As part of the high bandwidth interface 124, the general purpose media processor 12 integrates several fast communication channels 156 (FIG. 13) to communicate externally. These fast communication channels 156 preferably couple to external caches 150, which serve as a buffer to memory interfaces 152 coupled to standard memory 154. The caches 150 preferably comprise synchronous static random access memory ("SRAM"), each of which are sixty-four kilobytes in size; and the standard memories 154 comprise DRAM's. The memory interfaces 152 transmit data between the caches 150 and the standard memories 154. The standard memories 154 together form the main external memory for the general purpose media processor 12. The cache 150, memory interface 152, standard memory 154 and input/output channel 156 therefore make up a single external memory unit 158 for the general purpose media processor 12.

According to the presently preferred embodiment of the invention, the memory interface protocol embeds read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets preferably include check codes that will detect single-bit transmission errors and some multiple-bit errors. As many as eight operations may be in progress at a time in each external memory unit 158.

As shown in FIG. 13, up to four external memory units 158 may be cascaded together to expand the memory available to the general purpose media processor 12, and to improve the bandwidth of the external memory. Through such cascaded memory units 158, the memory interface 152 provides for the direct connection of multiple banks of standard memory 154 to maintain operation of the general purpose media processor 12 at sustained peak bandwidths.

According to one embodiment shown in FIG. 13, up to four standard memory devices 154 can be coupled to each memory interface 152. Each standard memory 154 thus includes as many as four banks of DRAM, each of which is preferably sixteen bits wide. The standard memories 154 are connected in parallel to the memory interface 152 forming a 72-bit wide data bus 160, where 64 bits are preferably provided for data transfer and eight bits are provided for error correction. In addition to the data bus 160, an address/control bus 162 is coupled between the memory interface 152 and each standard memory 154. The address/control bus 162 preferably comprises at least twelve address lines (4 kilobits x 16 memory size) and four control lines as shown in FIG. 13. An alternate manner for coupling the DRAM's to the memory interface 152 is illustrated in FIG. 14. As shown in FIG. 14, two banks of four DRAM single in-line memory modules are coupled in parallel to the memory interface 152. The memory interface 152 also supports interleaving to enhance bandwidth, and page mode accesses to improve latency for localized addressing.

Using standard DRAM components, the external memory units 158 achieve bandwidths of approximately two gigabits/second with the standard memories 154. When four such external memory units 158 are coupled via the communication channel 156, therefore, the total bandwidth of the external main memory system increases to one gigabyte/second. As discussed further below, in implementations with two or eight communication channels 156, the aggregate bandwidth increases to two and eight gigabytes/second, respectively.

A more detailed depiction of the communication channel 156 circuitry appears in FIG. 15. According to the preferred embodiment of the invention, each communication channel 156 comprises two unidirectional, byte-

wide, differential, packet-oriented data channels 156a, 156b (see FIG. 13). As explained above, where memory units 158 are cascaded together in series, the output of one memory unit 158 is connected to the input of another memory unit 158. The two unidirectional channels are thus connected through the memory units 158 forming a loop structure and make up a single bi-directional memory interface channel.

Referring to FIG. 15, each communication channel 156 is preferably eight bits wide, and each bit is transmitted differentially. For example, output transceiver 170 for bit D_{0out} transmits both D_0 and $/D_0$ signals over the communication channel 156. Additional transceivers are similarly provided for the remaining bits in the channel 156. (The transceiver 176 for bit D_{7out} and associated differential lines 178, 180 are shown in FIG. 15.) A CLK_{out} transceiver 182 is also provided to generate differential clock outputs 184, 186 over the channel 156. To complete the link between memory units 158, input transceivers 188-192 are provided in each memory unit 158 for each of the differential bits and clock signals transmitted over the communication channel 156. These input signals 172, 174, 178, 180, 184, 186 are preferably transmitted through input buffers 194-198 to other parts of the memory unit 158 (described above).

Each memory unit 158 also includes a skew calibrator 200 and phase locked loop ("PLL") 202. The skew calibrator 200 is used to control skew in signals output to the communication channel 156. Preferably, digital skew fields are employed, which include set numbers of delay stages to be inserted in the output path of the communication channel 156. Setting these fields, and the corresponding analog skew fields, permits a fine level of control over the relative skew between output channel signals.

The PLL 202 recovers the clock signal on either side of the communication channel 156 and is thus provided to remove clock jitter. The clock signals 184, 186 preferably comprise a single phase, constant rate clock signal. The clock signals 184, 186 thus contain alternating zero and one values transmitted with the same timing as the data signals 172, 174, 178, 180. The clock signal frequency is, therefore, one-half the byte data rate. The communication channel

156 preferably operates at constant frequency and contains no auxiliary control, handshaking or flow control information.

Each external memory unit 158 preferably defines two functional regions: a memory region, implemented by the cache 150 backed by standard memory 154 (see FIG. 13), and a configuration region, implemented by registers (not shown). Both regions are accessed by separate interfaces; the communication channel 156 is used to access the memory region, and a serial interface (described below) is used to access the configuration region. In the memory region, the caches 150 are preferably write-back (write-in) single-set (direct-map) caches for data originally contained in standard memory 154. All accesses to memory space should maintain consistency between the contents of the cache 150 and the contents of the standard memory 154. The configuration region registers provide the mechanism to detect and adjust skew in the communication channel 156. Software is preferably employed to adaptively adjust the skew in the channel 156 through digital skew fields, as explained above. The serial interface thus is used to configure the external memory units 158, set diagnostic modes and read diagnostic information, and to enable the use of a high-speed tester (not shown).

One presently preferred embodiment of the invention employs two byte-wide packet communication channels 156 (FIG. 16(a)). In order to further increase the bandwidth of the general purpose media processor 12, up to sixteen byte-wide packet communication channels 156 can be employed. Referring to FIG. 16(b), twelve communication channels, comprising eight memory channels 210, a ninth channel for parallel processing 212 (described below), and three input/output ("I/O") channels 214, are shown. Each of the communication channels 210-214 preferably employs the cascade configuration of four channel interface devices 216. (Each channel interface device 216 coupled to the memory channels 210 corresponds to the external memory unit 158 shown in FIG. 13.) Through each of the twelve communication channels shown in FIG. 16(b), the general purpose media processor 12 can request or issue read or write transactions. When not interleaved, the twelve channels provide a single contiguous memory space for each channel interface device 216.

Alternatively, memory accesses may be interleaved in order to provide for continuous access to the external memory system at the maximum bandwidth for the DRAM memories. In an interleaved configuration, at any point in time some memory devices will be engaged in row pre-charge, while others may be driving or receiving data, or receiving row or column addresses. The memory interface 152 (FIG. 13) thus preferably maps between a contiguous address space and each of the separate address spaces made available within each external memory unit 158. For maximum performance, therefore, the memory interface is interleaved so that references to adjacent addresses are handled by different memory devices. Moreover, in the preferred embodiment, additional memory operations may be requested before the corresponding DRAM bank is available. In an interleaved approach, these operations are placed in a queue until they can be processed. According to the preferred embodiment, memory writes have lower priority than memory reads, unless an attempt is made to read an address that is queued for a write operation. As those skilled in the art will appreciate, the depth of the memory write queue is dictated by the specific implementation.

Although up to four external memory units 158 are preferably cascaded to form effectively larger memories, some amount of latency may be introduced by the cascade. Packets of data transmitted over the communication channel 156 are uniquely addressed to a particular channel interface device 216. A packet received at a particular device, which specifies another module address, is automatically passed to the correct channel interface device 216. Unless the module address matches a particular device 216, that packet simply passes from the input to the output of the interface device 216. This mechanism divides the serial interconnection of interface devices 216 into strings, which function as a single larger memory or peripheral, but with possibly longer response latency.

In addition to the memory channels 210, the general purpose media processor 12 provides several communication channels 214 for communication with external input/output devices. Referring to FIG. 16(b), three input/output channels 214 having SRAM buffered memory (see FIG. 13) provide an interface

to external standard I/O devices (not shown). Like the eight memory channels 210, the three I/O channels 214 are byte-wide input/output channels intended to operate at rates of at least one gigahertz. The three I/O channels 214 also operate as a packet communication link to synchronous SRAM memory 208 within the channel interface device 216. A controller 226 within the channel interface device 216 completes the interface to the I/O devices.

The three I/O channels 214 preferably function in like manner to the memory channels 210 described above. The interface protocol for the three I/O channels 214 divides read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets also include a check code that will detect single-bit transmission errors and some multiple-bit errors. According to the preferred embodiment of the invention, as many as eight operations may progress in each interface device 216 at a time. As shown in FIG. 16(b), up to four channel interface devices 216 can be cascaded together to expand the bandwidth in the three I/O channels 214. A bit-serial interface (not shown) is also provided to each of the channel interface devices 216 to allow access to configuration, diagnostic and tester information at standard TTL signal levels at a more moderate data rate. (A more detailed description of the serial interface is provided below).

Like the memory channels 210, each I/O channel 214 includes nine signals -- one clock signal and eight data signals. Differential voltage levels are preferably employed for each signal. Each channel interface device 216 is preferably terminated in a nominal 50 ohm impedance to ground. This impedance applies for both inputs and outputs to the communication channel 156. A programmable termination impedance is preferred.

Interface Communication

According to one presently preferred embodiment of the invention, the channel interface devices 216 can operate as either master devices or slave devices. A master device is capable of generating a request on the communication channel 156 and receiving responses from the communication channel 156. Slave

devices are capable of receiving requests and generating responses, over the communication channel 156. A master device is preferably capable of generating a constant frequency clock signal and accepting signals at the same clock frequency over the communication channel 156. A slave device, therefore, should operate at the same clock rate as the communication channel 156, and generate no more than a specified amount of variation in output clock phase relative to input clock phase. The master device, however, can accept an arbitrary input clock phase and tolerates a specified amount of variation in clock phase over operating conditions.

Packets of information sent over the communication channel 156 preferably contain control commands, such as read or write operations, along with addresses and associated data. Other commands are provided to indicate error conditions and responses to the above commands. When the communication channel 156 is idle, such as during initialization and between transmitted packets, an idle packet, consisting of an all-zero byte and an all- one byte is transmitted through the communication channel 156. Each non-idle packet consists of two bytes or a multiple of two bytes, and begins with a byte having a value other than all zeros. All packets transmitted over the communication channel 156 also begin during a clock period in which the clock signal is zero, and all packets preferably end during a clock period in which the clock signal is one. A depiction of the preferred packet protocol format for transmission over the communication channel 156 appears in FIG. 17.

The general form of each packet is an array of bytes preferably without a specific byte ordering. The first byte contains a module address 250 ("ma") in the high order two bits; a packet identifier, usually a command 252 ("com"), in the next three bit positions; and a link identification number 254 ("lid") in the last three bit positions. The interpretation of the remaining bytes of a packet depend upon the contents of the packet identifier. The length of each packet is preferably implied by the command specified in the initial byte of the packet. A check byte is provided and computed as odd bit-wise parity with a leftward circular rotation after accumulating each byte. This technique provides

detection of all single-bit and some multiple-bit errors. but no correction is provided.

The modular address 250 field of each packet is preferably a two-bit field and allows for as many as four slave devices to be operated from a single communication channel 156. Module address values can be assigned in one of two fashions: either dynamically assigned through a configuration register (not shown), or assigned via static/geometric configuration pins. Dynamic assignment through a configuration register is the presently preferred method for assigning module address values.

The link identification number 254 field is preferably 3-bits wide and provides the opportunity for master devices to initiate as many as eight independent operations at any one time to each slave device. Each outstanding operation requires a distinct link identification number, but no ordering of operations should be implied by the value of the link identification field. Thus, there is preferably no requirement for link identification values 254 to be sequentially assigned either in requests or responses.

The receipt of packets over the communication channel 156 that do not conform to the channel protocol preferably generates an error condition. As those skilled in the art will appreciate, the level or degrees to which a specific implementation detects errors is defined by the user. In one presently preferred embodiment of the invention, all errors are detected, and the following protocol is employed for handling errors. For each error detected, the channel interface device 216 causes a response explicitly indicating the error condition. Channel interface devices 216 reporting an invalid packet will then suppress the receipt of additional packets until the error is cleared. The transmitted packet is otherwise ignored. However, even though the erroneous packet is ignored, the channel interface devices 216 preferably continue to process valid packets that have already been received and generate responses thereto. An identification of the presently preferred commands 252 to be used over the communication channel 156 are listed in FIG. 17.

In the master/slave preferred embodiment, the channel interface devices 216 forward packets that are intended for other devices connected to the communication channel 156. as described above. In slave devices, forwarding is performed based on the module address 250 field of the packet. Packets which contain a module address 250 other than that of the current device are forwarded on to the next device. All non-idle packets are thus forwarded including error packets. In master devices, forwarding is performed based on the link identifier number 254 of the packet. Packets that contain link identifier numbers 254 not generated by the specific channel interface device 216 are forwarded. In order to reduce transmission latency, a packet buffer may be provided. As those skilled in the art appreciate, the suitable size for the packet buffer depends on the amount of latency tolerable in a particular implementation.

A variety of master/slave ring configurations are possible using the high bandwidth interface 124 of the invention. Five ring configurations are currently preferred: single-master, dual-master, multiple-master, single-slave and multiple-master/multiple-slave. The simplest ring configuration contains a single non-forwarding master device and a single non-forwarding slave device. No forwarding is required for either device in this configuration as packets are sent directly to the recipient. A single-master ring, however, may contain a cascade of up to four slave devices (see FIGS. 13, 16). In the single-master ring configuration, each slave device is configured to a distinct module address, and each slave device forwards packets that contain module address fields unequal to their own. As discussed above, a single-master ring provides a larger memory or I/O capacity than a master-slave pair, but also introduces a potentially longer response latency. In the single-master ring, each slave device may have as many as eight transactions outstanding at any time, as described above.

The remaining combinations share many of the above basic attributes. In a dual-master pair. each master device may initiate read and write operations addressed to the other, and each may have up to eight such transactions outstanding. No forwarding is required for either device because packets are sent directly to the recipient. A multiple-master ring may contain multiple master

devices and a single slave device. In this configuration, the slave device need not forward packets as all input packets are designated for the single slave device. A multiple-master ring may contain multiple master devices and as many as four slave devices. Each slave device may have up to eight transactions outstanding, and each master device may use some of those transactions. In a preferred embodiment, a master also has the capability to detect a time-out condition or when a response to a request packet is not received. Further aspects of inter-processor communications and configurations are discussed below in connection with FIG. 18.

Serial Bus

In one preferred embodiment of the invention, the general purpose media processor 12 includes a serial bus (not shown). The serial bus is designed to provide bootstrap resources, configuration, and diagnostic support to the general purpose media processor 12. The serial bus preferably employs two signals, both at TTL levels, for direct communication among many devices. In the preferred embodiment, the first signal is a continuously running clock, and the second signal is an open-collector bi-directional data signal. Four additional signals provide geographic addresses for each device coupled to the serial bus. A gateway protocol, and optional configurable addressing, each provide a means to extend the serial bus to other buses and devices. Although the serial bus is designed for implementation in a system having a general purpose media processor 12, as those skilled in the art will appreciate, the serial bus is applicable to other systems as well.

Because the serial bus is preferably used for the initial bootstrap program load of the general purpose media processor 12, the bootstrap ROM is coupled to the serial bus. As a result, the serial bus needs to be operational for the first instruction fetch. The serial bus protocol is therefore devised so that no transactions are required for initial bus configuration or bus address assignment.

According to the preferred embodiment, the clock signal comprises a continuously running clock signal at a minimum of 20 megahertz. The amount

of skew, if any, in the clock signal between any two serial bus devices should be limited to be less than the skew on the data signal. Preferably, the serial data signal is a non-inverted open collector bi-directional data signal. TTL levels are preferred for communication on the serial bus. and several termination networks may be employed for the serial data signal. A simple preferred termination network employs a resistive pull-up of 220 ohms to 3.3 volts above V_{cc}. An alternate embodiment employs a more complex termination network such as a termination network including diodes or the "Forced Perfect Termination" network proposed for the SCSI-2 standard, which may be advantageous for larger configurations.

The geographic addressing employed in the serial bus is provided to insure that each device is addressable with a number that is unique among all devices on the bus and which also preferably reflects the physical location of the device. Thus, the address of each device remains the same each time the system is operated. In one preferred embodiment, the geographic address is composed of four bits, thus allowing for up to 16 devices. In order to extend the geographic addressing to more than 16 devices, additional signals may be employed such as a buffered copy of the clock signal or an inverted copy of the clock signal (or both).

The serial bus preferably incorporates both a bit level and packet protocol. The bit level protocol allows any device to transmit one bit of information on the bus, which is received by all devices on the bus at the same time. Each transmitted bit begins at the rising edge of the clock signal and ends at the next rising edge. The transmitted bit value is sampled at the next rising edge of the clock signal. According to one preferred embodiment where the serial data signal is an open collector signal, the transmission of a zero bit value on the bus is achieved by driving the serial data signal to a logical low value. In this embodiment, the transmission of a one bit value is achieved by releasing the serial data signal to obtain a logical high value. If more than one device attempts to transmit a value on the same clock, the resulting value is a zero if any device transmits a zero value, and one if all devices transmit a one value. This provides a "wired-AND" collision mechanism. as those skilled in the art will appreciate. If

two or more devices transmit the same value on the same clock cycle, however, no device can detect the occurrence of a collision. In such cases, the transaction, which may occur frequently in some implementations, preferably proceeds as described below.

The packet protocol employed with the serial bus uses the bit level protocol to transmit information in units of eight bits or multiples of eight bits. Each packet transmission preferably begins with a start bit in which the serial data signal has a zero (driven) value. After transmitting the eight data bits, a parity bit is transmitted. The transmission continues with additional data. A single one (released) bit is transmitted immediately following the least significant bit of each byte signaling the end of the byte.

On the cycle following the transmission of the parity bit, any device may demand a delay of two cycles to process the data received. The two cycle delay is initiated by driving the serial data signal (to a zero value) and releasing the serial data signal on the next cycle. Before releasing the serial data signal, however, it is preferable to insure that the signal is not being driven by any other device. Further delays are available by repeating this pattern.

In order to avoid collisions, a device is not permitted to start a transmission over the serial bus unless there are no currently executing transactions. To resolve collisions that may occur if two devices begin transmission on the same cycle, each transmitting device should preferably monitor the bus during the transmission of one (released) bits. If any of the bits of the byte are received as zero when transmitting a one, the device has lost arbitration and must cease transmission of any additional bits of the current byte or transaction.

According to the preferred embodiment of the invention, a serial bus transaction consists of the transmission of a series of packets. The transaction begins with a transmission by the transaction initiator, which specifies the target network, device, length, type and payload of the transaction request. The transaction terminates with a packet having a type field in a specified range. As a result, all devices connected to the serial bus should monitor the serial data signal

to determine when transactions begin and end. A serial bus network may have multiple simultaneous transactions occurring, however, so long as the target and initiator network addresses are all disjoint.

Parallel Processing

In one preferred embodiment of the invention, two or more general purpose media processors 12 can be linked together to achieve a multiple processor system. According to this embodiment, general purpose media processors 12 are linked together using their high bandwidth interface channels 124, either directly or through external switching components (not shown). The dual-master pair configuration described above can thus be extended for use in multiple-master ring configurations. Preferably, internal daemons provide for the generation of memory references to remote processors, accesses to local physical memory space, and the transport of remote references to other remote processors. In a multi-processor environment, all general purpose media processors 12 run off of a common clock frequency. as required by the communication channels 156 that connect between processors.

Referring to FIG. 18, each general purpose media processor 12 preferably includes at least a pair of inter-processor links 218 (see also FIG. 16(b)). In one configuration. both pairs of inter-processor links 218 can be connected between the two processors 12 to further enhance bandwidth. As shown in FIG. 18(a) several processors 12 may be interconnected in a linear network employing the transponder daemons in each processor. In an alternate embodiment shown in FIG. 18(b), the inter-processor links 218 may be used to join the general purpose media processors 12 in a ring configuration. Alternatively still, general purpose media processors 12 may be interconnected into a two-dimensional network of processors of arbitrary size, as shown in FIG. 18(c). Sixteen processors are connected in FIG. 18(c) by connecting four ring networks. In yet another alternate embodiment, by connecting the inter-processor links 218 to external switching devices (not shown), multi-processors with a large

number of processors can be constructed with an arbitrary interconnection topology.

The requester, responder and transponder daemons preferably handle all inter-processor operations. When one general purpose media processor 12 attempts a load or store to a physical address of a remote processor, the requester daemon autonomously attempts to satisfy the remote memory reference by communicating with the external device. The external device may comprise another processor 12 or a switching device (not shown) that eventually reaches another processor 12. Preferably, two requester daemons are provided each processor 12, which act concurrently on two different byte channels and/or module addresses. The responder daemon accepts writes from a specified channel and module address, which enables an external device to generate transaction requests in local memory or to generate processor events. The responder daemon also generates link level writes to the same external device that communicated responses for the received transaction request. Two such responder daemons are preferably provided; each of which operate concurrently to two different byte channels and/or module addresses.

The transponder daemon accepts writes from a specified channel and module address, which enable an external device to cause a requester daemon to generate a request on another channel and module address. Preferably, two such transponder daemons are provided, each of which act concurrently (back-to-back) between two different byte channel and/or module addresses. As those skilled in the art will appreciate, the requester, responder and transponder daemons must act cooperatively to avoid deadlock that may arise due to an imbalance of requests in the system. Deadlocks prevent responses from being routed to their destinations, which may defeat the benefits of a multi-processor distributed system.

According to one presently preferred embodiment of the invention, the general purpose media processor 12 can be implemented as one or more integrated circuit chips. Referring to FIG. 19, the presently preferred embodiment of the general purpose media processor 12 consists of a four-chip set. In the four-chip set, a general purpose media processor 12 is manufactured as a stand alone

integrated circuit. The stand alone integrated circuit includes a memory management unit 122, instruction and data cache/buffers 118, 120, and an execution unit 100. A plurality of signal input/output pads 260 are provided around the circumference of the integrated circuit to communicate signals to and from the general purpose media processor 12 in a manner generally known in the art.

The second and third chips of the four-chip set comprise in an external memory element 158 and a channel interface device 216. The external memory element 158 includes an interface to the communication channel 156, a cache 150 and a memory interface 152. The channel interface device 216 also includes an interface to the communication channel 156, as well as buffer memory 262, and input/output interfaces 264. Both the external memory element 158 and the channel interface device 216 include a plurality of input/output signal pads 260 to communicate signals to and from these devices in a generally known manner.

The fourth integrated circuit chip comprises a switch 226, which allows for installation of the general purpose media processor 12 in the heterogeneous network 38. In addition to the plurality of input/output pads 260, the switch 226 includes an interface to the communication channel 156. The switch 226 also preferably includes a buffer 262, a router 266, and a switch interface 268.

As those skilled in the art will appreciate, many implementations for the general purpose media processor 12 are possible in addition to the four-chip implementation described above. Rather than an integrated approach, the general purpose media processor can be implemented in a discrete manner. Alternatively, the general purpose media processor 12 can be implemented in a single integrated circuit, or in an implementation with fewer than four integrated circuit chips. Other combinations and permutations of these implementations are contemplated.

There has been described a system for processing streams of media data at substantially peak rates to allow for real time communication over a large heterogeneous network. The system includes a media processor at its core that is capable of processing such media data streams. The heterogeneous network

consists of, for example, the fiber optic/coaxial cable/twisted wire network in place throughout the U.S. To provide for such communication of media data, a media processor according to the invention is disposed at various locations throughout the heterogeneous network. The media processor would thus function both in a server capacity and at an end user site within the network. Examples of such end user sites include televisions, set-top converter boxes, facsimile machines, wireless and cellular telephones, as well as large and small business and industrial applications.

To achieve such high rates of data throughput, the media processor includes an execution unit, high bandwidth interface, memory management unit, and pipelined instruction and data paths. The high bandwidth interface includes a mechanism for transmitting media data streams to and from the media processor at rates at or above the gigahertz frequency range. The media data stream can consist of transmission, presentation and storage type data transmitted alone or in a unified manner. Examples of such data types include audio, video, radio, network and digital communications. According to the invention, the media processor is dynamically partitionable to process any combination or permutation of these data types in any size.

A programmable, general purpose media processor system presents significant advantages over current multimedia communications. Rather than rigid, costly and inefficient specialized processors, the media processor provides a general purpose instruction set to ease programmability in a single device that is capable of performing all of the operations of the specialized processor combination. Providing a uniform instruction set for all media related operations eliminates the need for a programmer to learn several different instruction sets, each for a different specialized processor. The complexity of programming the specialized processors to work together and communicate with one another is also greatly reduced. The unified instruction set is also more efficient. Highly specialized general calculation instructions that are tailored to general or special types of calculations rather than enhancing communication are eliminated.

Moreover, the media processor system can be easily reprogrammed simply by transmitting or downloading new software over the network. In the specialized processor approach, new programming usually requires the delivery and installation of new hardware. Reprogramming the media processor can be done electronically, which of course is quicker and less costly than the replacement of hardware.

It is to be understood that a wide range of changes and modifications to the embodiments described above will be apparent to those skilled in the art and are contemplated. It is therefore intended that the foregoing detailed description be regarded as illustrative rather than limiting, and that it be understood that it is the following claims, including all equivalents, that are intended to define the spirit and scope of this invention.

Set forth on the following pages (46-406) is a more detailed discussion of the operations, and the functionality of the general purpose media processor 12.

Introduction

MicroUnity's Terpsichore System Architecture describes general-purpose processor, memory, and interface subsystems, organized to operate at enormously higher bandwidth rates than traditional computers.

Terpsichore's Euterpe processor performs integer, floating point, and signal processing operations at data rates up to 512 bits (i.e., up to four 128-bit operand groups) per instruction. The instruction set design carries the concept of streamlining beyond Reduced Instruction Set Computer (RISC) architectures, since it targets implementations that issue several instructions per machine cycle.

The Terpsichore memory subsystem provides 64-bit virtual and physical addressing for UNIX, Mach, and other advanced OS environments. Caches supply the high data and instruction issue rates of the processor, and support coherency primitives for scalable multiprocessors. The memory subsystem includes mechanisms for sustaining high data rates not only in block transfer modes, but also in non-unit stride and scatter/gather access patterns.

Hermes channels provide 64-bit transfers between subsystem components with gigabyte-per-second bandwidth. Terpsichore's Cerberus serial bus provides a flexible, robust and inexpensive mechanism to handle system initialization, configuration, availability, and error recovery. Mnemosyne memory interface devices provide for the integration of large numbers of industry-standard memory components into Terpsichore systems. Persephone devices enable Terpsichore systems to utilize industry-standard PCI interface cards.

Terpsichore's Calliope interface subsystem is tightly integrated with the processor and memory, to supply both the bandwidth and real-time response needs of video, audio, network, and mass storage interfaces. Integration provides for the sharing of memory bandwidth among these devices and the processor, without distributed or dedicated buffer memories in each interface adapter.

Terpsichore's Euterpe processor incorporates Icarus interprocessor interfaces for assembly of small-scale, coherently-cached, shared-memory multiprocessors, without additional circuitry. Icarus interfaces may also be used to connect Terpsichore processors to a high-performance switching fabric for large-scale multiprocessors, or to adapters to standard interprocessor interfaces, such as Scalable Coherent Interface (IEEE standard 1596-1992).

The goal of the Terpsichore architecture is to integrate these processor, memory, and interface capabilities with optimal simplicity and generality. From the software perspective, the entire machine state consists of a program counter, a single bank of 64 general-purpose 64-bit registers, and a linear byte-addressed shared memory space with mapped interface registers. All interrupts and exceptions are precise, and occur with low overhead.

This document is intended for Terpsichore software and hardware developers alike, and defines the interface at which their designs must meet. Terpsichore pursues the most efficient tradeoffs between hardware and software complexity

by making all processor, memory, and interface resources directly accessible to high-level language programs.

Conformance

To ensure that Terpsichore systems are able to freely interchange data, user-level programs, system-level programs and interface devices, the Terpsichore system architecture reaches above the processor level architecture.

Mandatory and Optional Areas

A computer system conforms to the requirements of the Terpsichore System Architecture if and only if it implements all the specifications described in this document and other specifications included by reference. Conformance to the specification is mandatory in all areas, including the instruction set, memory management system, interface devices and external interfaces, and bootstrap ROM functional requirements, except where explicit options are stated.

Optional areas include:

Number of processor threads
Size of first-level cache memories
Existence of a second-level cache
Size of second-level cache memory
Size of system-level memory
Existence of certain optional interface device interfaces

Conformance to the specification is also optional regarding the physical implementation of internal interfaces, specifically that of the Cerberus serial bus architecture, the Hermes high-bandwidth channel architecture, and the Icarus interprocessor interconnection architecture. An implementation may replace, modify or eliminate these interfaces, provided that the software-level functionality is unchanged.

Upward-compatible Modifications

From time to time, MicroUnity may modify the architecture in an upward-compatible manner, such as by the addition of new instructions, definition of reserved bits in system state, or addition of new standard interfaces. Such modifications will be added as options, so that designs which conform to this version of the architecture will conform to future, modified versions.

Additional devices and interfaces, not covered by this standard may be added in specified regions of the physical memory space, provided that system reset places these devices and interfaces in an inactive state that does not interfere with the operation of software that runs in any conformant system. The software interface requirements of any such additional devices and interfaces must be made as widely available as this architecture specification.

Promotion of Optional Features

It is most strongly recommended that such optional instructions, state or interfaces be implemented in all conforming designs. Such implementations enhance the value of the features in particular and the architecture as a whole by broadening the set of implementations over which software may depend upon the presence of these features.

Implementations which fail to implement these features may encounter unacceptable levels of overhead when attempting to emulate the features by exception handlers or use of virtual memory. This is a particular concern when involved in code which has real-time performance constraints.

In order that upward-compatible optional extensions of the original Terpsichore system architecture may be relied upon by system and application software, MicroUnity may upon occasion promote optional features to mandatory conformance for implementations designed or produced after a suitable delay upon such notification by publication of future version of the specification.

Unrestricted Physical Implementation

Nothing in this specification should be construed to limit the implementation choices of the conformant system beyond the specific requirements stated herein. In particular, a computer system may conform to the Terpsichore System Architecture while employing any number of components, dissipate any amount of heat, require any special environmental facilities, or be of any physical size.

Draft Version

This document is a draft version of the architectural specification. In this form, conformance to this document may not be claimed or implied. MicroUnity may change this specification at any time, in any manner, until it has been declared final. When this document has been declared final, the only changes will be to correct bugs, defects or deficiencies, and to add upward-compatible optional extensions.

Common Elements

Notation

The descriptive notation used in this document is summarized in the table below:

x + y	two's complement or floating-point addition of x and y. Result is the same size as the operands, and operands must be of equal size.
x - y	two's complement or floating-point subtraction of y from x. Result is the same size as the operands, and operands must be of equal size.
x * y	two's complement or floating-point multiplication of x and y. Result is the same size as the operands, and operands must be of equal size.
x / y	two's complement or floating-point division of x by y. Result is the same size as the operands, and operands must be of equal size.
x = y	two's complement or floating-point equality comparison between x and y. Result is a single bit, and operands must be of equal size.
x ≠ y	two's complement or floating-point inequality comparison between x and y. Result is a single bit, and operands must be of equal size.
x < y	two's complement or floating-point less than comparison between x and y. Result is a single bit, and operands must be of equal size.
x ≥ y	two's complement or floating-point greater than or equal comparison between x and y. Result is a single bit, and operands must be of equal size.
$\sqrt{x}$	floating-point square root of x
x ll y	concatenation of bit field x to left of bit field y
x^y	binary digit x repeated, concatenated y times. Size of result is y.
x_y	extraction of bit y (using little-endian bit numbering) from value x. Result is a single bit.
$x_{y..z}$	extraction of bit field formed from bits y through z of value x
x?y:z	value of y, if x is true, otherwise value of z. Value of x is a single bit.
x ← y	bitwise assignment of x to value of y
Sn	signed, two's complement, binary data format of n bytes
Un	unsigned binary data format of n bytes
Fn	floating-point data format of n bytes

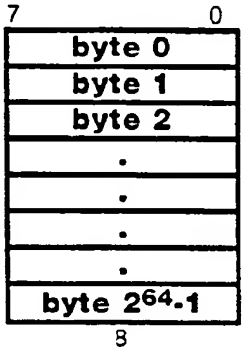
descriptive notation

Bit ordering

The ordering of bits in this document is always little-endian, regardless of the ordering of bytes within larger data structures. Thus, the least-significant bit of a data structure is always labeled 0 (zero), and the most-significant bit is labeled as the data structure size (in bits) minus one.

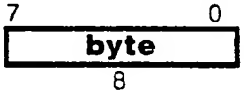
Memory

Terpsichore memory is an array of 2^{64} bytes, without a specified byte ordering, which is physically distributed among various Terpsichore components.



Byte

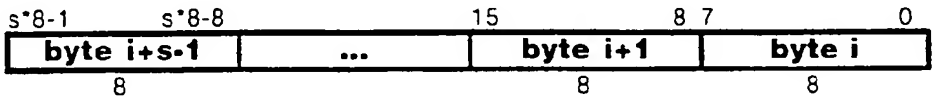
A byte is a single element of the memory array, consisting of 8 bits:



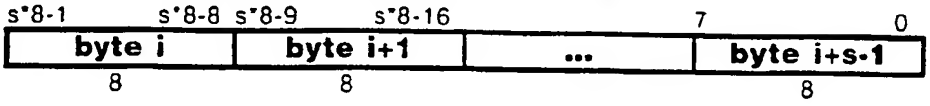
Byte ordering

Larger data structures are constructed from the concatenation of bytes in either little-endian or big-endian byte ordering. A memory access of a data structure of size s at address i is formed from memory bytes at addresses i through i+s-1. Unless otherwise specified, there is no specific requirement of alignment: it is not generally required that i be a multiple of s. Aligned accesses are preferred whenever possible, however, as they will often require one less processor or memory clock cycle than unaligned accesses.

With little-endian byte ordering, the bytes are arranged as:



With big-endian byte ordering, the bytes are arranged as:



Terpsichore memory is byte-addressed, using either little-endian or big-endian byte ordering. For consistency with the bit ordering, and for compatibility with x86 processors, Terpsichore uses little-endian byte ordering when an ordering must be selected. Euterpe load and store instructions are available for both little-endian and big-endian byte ordering. The selection of byte ordering is dynamic, so that little-endian and big-endian processes, and even data structures within a process, can be intermixed on the processor.

Memory read/load semantics

Terpsichore memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory read must have no side-effects on the contents of the addressed memory nor on the contents of any other memory.

Memory write/store semantics

Terpsichore memory, including memory-mapped registers, must conform to the following requirements regarding side-effects of read or load operations:

A memory write must have no side-effects on the contents of the addressed memory. A memory write may cause side-effects on the contents of memory not addressed by the write operation, however, a second memory write of the same value to the same address must have no side-effects on any memory; memory write operations must be idempotent.

Euterpe store instructions which are weakly ordered may have side-effects on the contents of memory not addressed by the store itself; subsequent load instructions which are also weakly ordered may or may not return values which reflect the side-effects.

Data

Euterpe provides eight-byte (64-bit) virtual and physical address sizes, and eight-byte (64-bit) and sixteen-byte (128-bit) data path sizes, and uses fixed-length four-byte (32-bit) instructions. Arithmetic is performed on two's-complement or unsigned binary and ANSI/IEEE standard 754-1985 conforming binary floating-point number representations.

Fixed-point Data

Bit

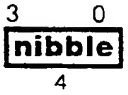
A bit is a primitive data element:

Peck

A peck is the catenation of two bits:

Nibble

A nibble is the catenation of four bits:

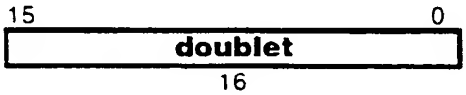


Byte

A byte is the catenation of eight bits, and is a single element of the memory array:

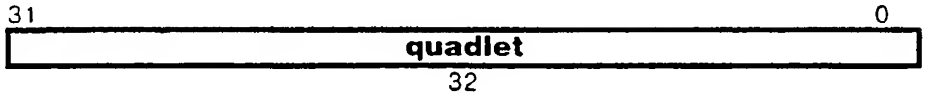
Doublet

A doublet is the catenation of 16 bits, and is the concatenation of two bytes:



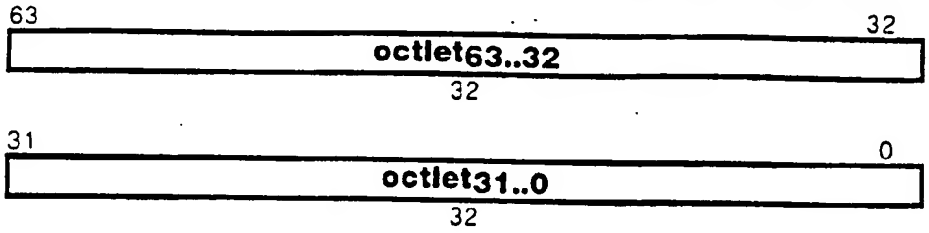
Quadlet

A quadlet is the catenation of 32 bits, and is the concatenation of four bytes:



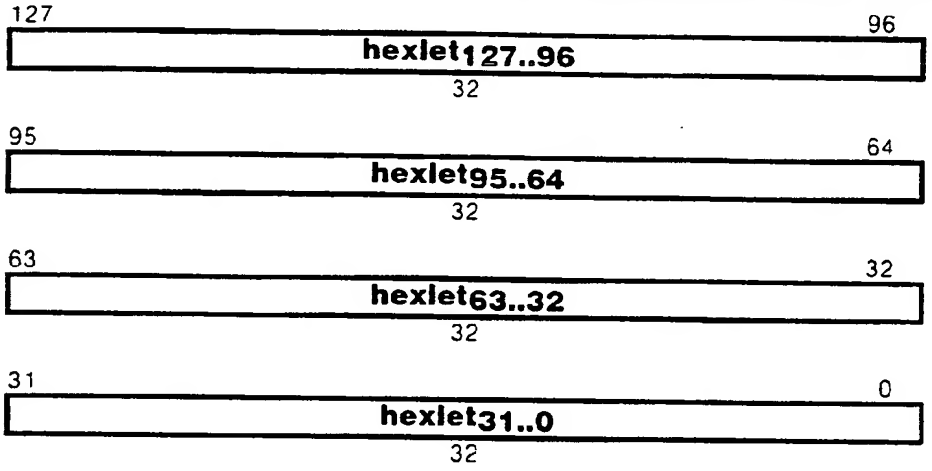
Octlet

An octlet is the catenation of 64 bits, and is the concatenation of eight bytes:



Hexlet

A hexlet is the catenation of 128 bits, and is the concatenation of sixteen bytes:



Address

Terpsichore addresses are octlet quantities.

Floating-point Data

Terpsichore's floating-point formats are designed to satisfy ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic. Standard 754 leaves certain aspects to the discretion of the implementor:

Terpsichore adds additional half-precision and quad-precision formats to standard 754's single-precision and double-precision formats. Terpsichore's double-precision satisfies standard 754's precision requirements for a single-extended format, and Terpsichore's quad-precision satisfies standard 754's precision requirements for a double-extended format.

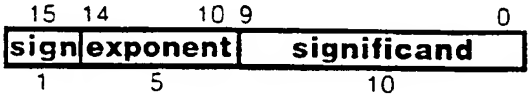
Quiet NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit cleared. Quiet NaN

values generated by default exception handling of standard operations have a zero sign bit, an exponent field of all one bits, and a significand field with the most significant bit cleared, the next-most significant bit set, and all other bits cleared.

Signaling NaN values are denoted by any sign bit value, an exponent field of all one bits, and a non-zero significand with the most significant bit set.

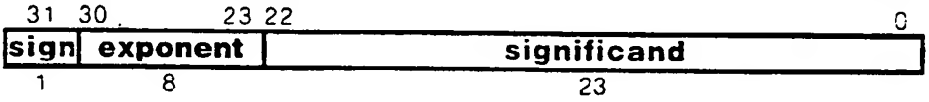
Half-precision Floating-point

Terpsichore half precision uses a format similar to standard 754's requirements, reduced to a 16-bit overall format. The format contains sufficient precision and exponent range to hold a 12-bit signed integer.



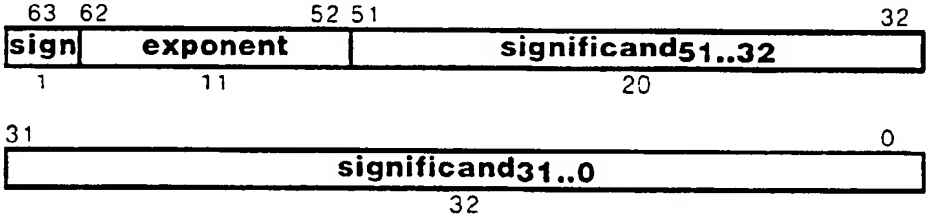
Single-precision Floating-point

Terpsichore single precision satisfies standard 754's requirements for "single."



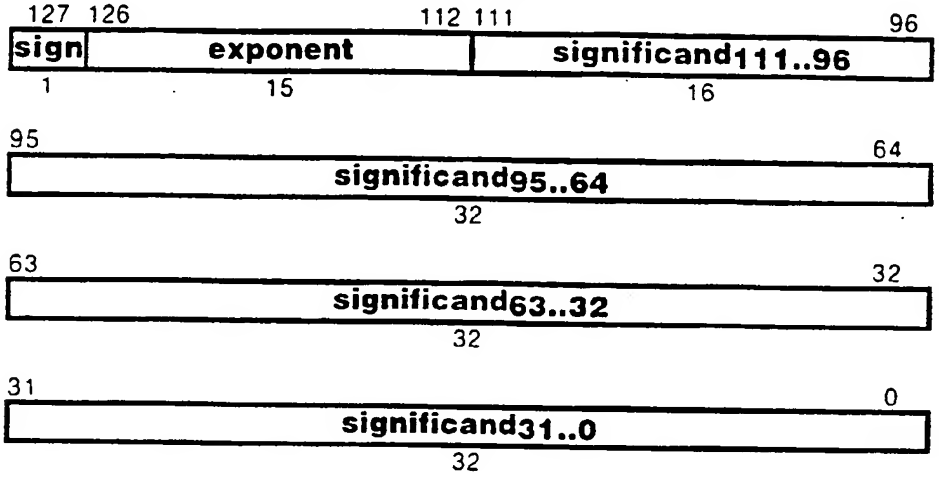
Double-precision Floating-point

Terpsichore double precision satisfies standard 754's requirements for "double."



Quad-precision Floating-point

Terpsichore quad precision satisfies standard 754's requirements for "double extended," but has additional significand precision to use 128 bits.



Euterpe Processor

MicroUnity's Euterpe processor provides the general-purpose, high-bandwidth computation capability of the Terpsichore system. Euterpe includes high-bandwidth data paths, register files, and a memory hierarchy. Euterpe's memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. Euterpe's interfaces include flash ROM, synchronous DRAM, Cerberus serial bus, Hermes high-bandwidth channels, and simple keyboard and display.

Architectural Framework

The Euterpe architecture builds upon MicroUnity's Hermes high-bandwidth channel architecture and upon MicroUnity's Cerberus serial bus architecture, and complies with the requirements of Hermes and Cerberus. Euterpe uses parameters A and W as defined by Hermes.

The Euterpe architecture defines a compatible framework for a family of implementations with a range of capabilities. The following implementation-defined parameters are used in the rest of the document in boldface. The value indicated is for MicroUnity's first Euterpe implementation.

Parameter	Interpretation	Value	Range of legal values
T	number of execution threads	5	$1 \le \mathbf{T} \le 31$
I	support for Icarus	0	$0 \le \mathbf{I} \le 1$
i	$\log_2$ Hermes words per interleave block	1	$0 \le \mathbf{i} \le 1$
H	number of Hermes channels	2	$0 \le \mathbf{H} \le 15$
$\mathbf{C_c}$	instruction SRAM can be all cache	0	$0 \le \mathbf{C_c} \le 1$
$\mathbf{C_b}$	instruction SRAM can be all buffer	1	$0 \le \mathbf{C_b} \le 1$
C	$\log_2$ cache blocks in instruction SRAM (cache+buffer)	9	$0 \le \mathbf{C} \le 31$
$\mathbf{D_c}$	data SRAM can be all cache	0	$0 \le \mathbf{D_c} \le 1$
$\mathbf{D_b}$	data SRAM can be all buffer	1	$0 \le \mathbf{D_b} \le 1$
D	$\log_2$ cache blocks in data SRAM (cache+buffer)	9	$0 \le \mathbf{D} \le 31$
L	$\log_2$ entries in local TLB	0	$0 \le \mathbf{L} \le 3$
G	$\log_2$ entries in global TLB	6	$0 \le \mathbf{G} \le 15$

Interfaces and Block Diagram

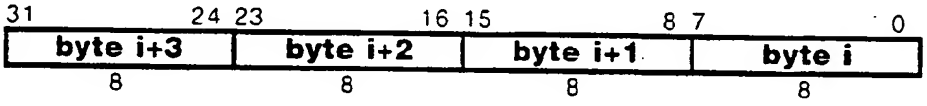
Instruction

Instruction Mnemonics

Instruction mnemonics are usually written with periods (.) separating elements of the mnemonic to make them easier to understand. Terpsichore assemblers and other code tools treat these periods as optional; the mnemonics are designed to be parsed uniquely either with or without the periods.

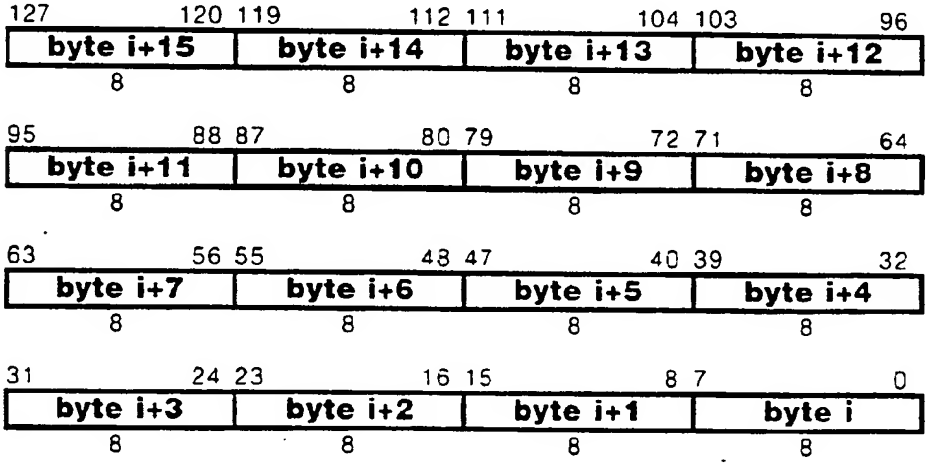
Instruction Structure

A Terpsichore instruction is specifically defined as a four-byte structure with the little-endian ordering shown below. It is different from the quadlet defined above because the placement of instructions into memory must be independent of the byte ordering used for data structures. Instructions must be aligned on four-byte boundaries: in the diagram below, i must be a multiple of 4.

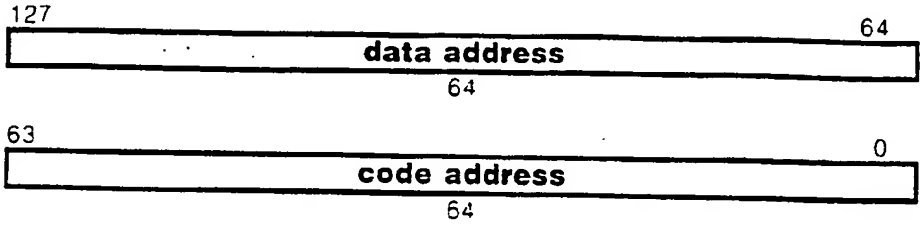


Gateway

A Terpsichore gateway is specifically defined as a 16-byte structure with the little-endian ordering shown below. A gateway contains a code address and a data address used to invoke a procedure at a higher privilege level securely. Gateways are marked by protection information specified in the TLB. Gateways must be aligned on 16-byte boundaries, that is, in the diagram below, i must be a multiple of 16.



The gateway contains two data items within its structure, a code address and a data address:

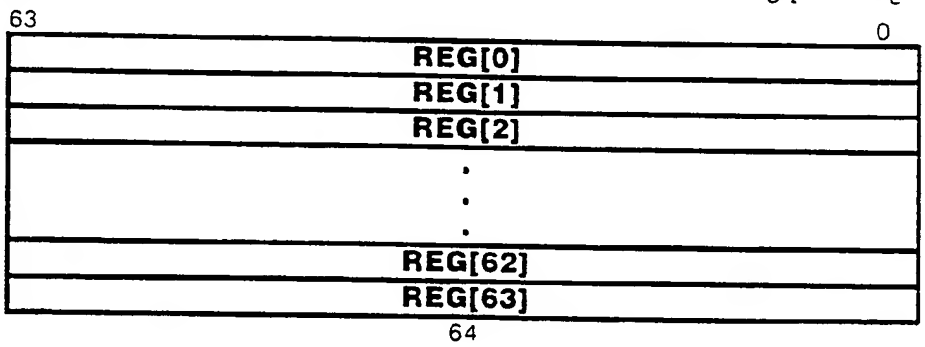


User state

The user state consists of hardware data structures that are accessible to all conventional compiled code. The Terpsichore user state is designed to be as regular as possible, and consists only of the general registers, the program counter, and virtual memory. There are no specialized registers for condition codes, operating modes, rounding modes, integer multiply/divide, or floating-point values.

General Registers

Terpsichore user state includes 64 general registers. All are identical; there is no dedicated zero-valued register, and there are no dedicated floating-point registers.



Definition

```
def val ← RegRead(rn, size)
    case size of
        64:
            val ← REG[rn]
        128:
            if rn0 then
                raise ReservedInstruction
            endif
            val ← REG[rn+1] || REG[rn]
    endcase
enddef
```

```
def RegWrite(rn, size, val)
    case size of
        64:
            REG[rn] ← val63..0
        128:
            if rn0 then
                raise ReservedInstruction
            endif
            REG[rn+1] ← val127..64
            REG[rn] ← val63..0
    endcase
enddef
```

Program Counter

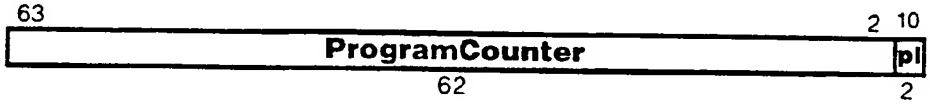
The program counter contains the address of the currently executing instruction. This register is implicitly manipulated by branch instructions, and read by branch instructions that save a return address in a general register.

Privilege Level

The privilege level register contains the privilege level of the currently executing instruction. This register is implicitly manipulated by branch gateway and branch down instructions, and read by branch gateway instructions that save a return address in a general register.

Program Counter and Privilege Level

The program counter and privilege level may be packed into a single octlet. This combined data structure is saved by the Branch Gateway instruction and restored by the Branch Down instruction.



System state

The system state consists of the facilities not normally used by conventional compiled code. These facilities provide mechanisms to execute such code in a fully virtual environment. All system state is memory mapped, so that it can be manipulated by compiled code.

Fixed-point

Terpsichore provides load and store instructions to move data between memory and the registers, branch instructions to compare the contents of registers and to transfer control from one code address to another, and arithmetic operations to perform computation on the contents of registers, returning the result to registers.

Load and Store

The load and store instructions move data between memory and the registers. When loading data from memory into a register, values are zero-extended or sign-extended to fill the register. When storing data from a register into memory, values are truncated on the left to fit the specified memory region.

Load and store instructions that specify a memory region of more than one byte may use either little-endian or big-endian byte ordering: the size and ordering are explicitly specified in the instruction. Regions larger than one byte may be either aligned to addresses that are an even multiple of the size of the region, or of unspecified alignment: alignment checking is also explicitly specified in the instruction.

The load and store instructions are used for fixed-point data as well as floating-point and digital signal processing data: Terpsichore has a single bank of registers for all data types.

Swap instructions provide multithread and multiprocessor synchronization, using indivisable operations: add-and-swap, compare-and-swap, and multiplex-and-swap. A store-multiplex operation provides the ability to indivisably write to a portion of an octlet. These instructions always operate on aligned octlet data, using either little-endian or big-endian byte ordering.

Branch Conditionally

The fixed-point compare-and-branch instructions provide all arithmetic tests for equality and inequality of signed and unsigned fixed-point values. Tests are performed either between two operands contained in general registers, or on the bitwise and of two operands. Depending on the result of the compare, either a branch is taken, or not taken. A taken branch causes an immediate transfer of the program counter to the target of the branch, specified by a 12-bit signed offset from the location of the branch instruction. A non-taken branch causes no transfer; execution continues with the following instruction.

Branch Unconditionally

Other branch instructions provide for unconditional transfer of control to addresses too distant to be reached by a 12-bit offset, and to transfer to a target while placing the location following the branch into a register. The branch through gateway instruction provides a secure means to access code at a higher privilege level, in a form similar to a normal procedure call.

Arithmetic Operations

The fixed-point arithmetic operations include add, subtract, multiply, divide, shifts, and set on compare, all using octlet-sized operands. Multiply and divide operations produce hexlet results; all other operations produce octlet results.

When specified, add, subtract, and shift operations may cause a fixed-point arithmetic exception to occur on resulting conditions such as signed overflow, or signed or unsigned equality or inequality to zero.

Addressing Operations

A subset of the arithmetic operations are available as addressing operations. These addressing operations may be performed at a point in the Euterpe processor pipeline so that they may be completed prior to or in conjunction with the execution of load and store operations in a "superspring" pipeline in which other arithmetic operations are deferred until the completion of load and store operations.

Floating-point

Terpsichore provides all the facilities mandated and recommended by ANSI/IEEE standard 754-1985: Binary Floating-point Arithmetic, with the use of supporting software.

Branch Conditionally

The floating-point compare-and-branch instructions provide all the comparison types required and suggested by the IEEE floating-point standard. These floating-point comparisons augment the usual types of numeric value comparisons with special handling for NaN (not-a-number) values. A NaN compares as "unordered" with respect to any other value, even that of an identical NaN.

Terpsichore floating-point compare-and-branch instructions do not generate an exception on comparisons involving quiet or signaling NaNs; if such exceptions are desired, they can be obtained by combining the use of a floating-point compare and set instruction, with either a floating-point compare-and-branch instruction on the floating-point operands or a fixed-point compare-and-branch on the set result.

Because the less and greater relations are anti-commutative, one of each relation that differs from another only by the replacement of an L with a G in the code can be removed by reversing the order of the operands and using the other code. Thus, a UL relation can be used in place of a UG relation by swapping the operands to the compare-and-branch or compare-and-set instruction.

The E and NE relations can be used to determine the unordered condition of a single operand by comparing the operand with itself.

The following floating-point compare-and-branch relations are provided:

Mnemonic		Branch taken if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
NE	!=	T	T	T	F	no	no
UE	?=	T	F	F	T	no	no
NUE	!?=	F	T	T	F	no	no
NUGE	!?>=	F	F	T	F	no	no
UGE	?>=	T	T	F	T	no	no
UL	?<	T	F	T	F	no	no
NUL	!?<	F	T	F	T	no	no

compare-and-branch relations

Compare-and-set

The floating-point compare-and-set instructions provide all the comparison types supported as compare-and-branch instructions. Terpsichore floating-point compare-and-set instructions may optionally generate an exception on comparisons involving quiet or signaling NaNs.

The following floating-point compare-and-branch relations are provided:

Mnemonic		Result if values compare as:				Exception if	
code	C-like	Unord-ered	Greater	Less	Equal	unord-ered	invalid
E	==	F	F	F	T	no	no
NE	!=	T	T	T	F	no	no
UE	?=	T	F	F	T	no	no
NUE	!?=	F	T	T	F	no	no
NUGE	!?>=	F	F	T	F	no	no
UGE	?>=	T	T	F	T	no	no
UL	?<	T	F	T	F	no	no
NUL	!?<	F	T	F	T	no	no
E.X	==	F	F	F	T	no	yes
NE.X	!=	T	T	T	F	no	yes
UE.X	?=	T	F	F	T	no	yes
NUE.X	!?=	F	T	T	F	no	yes
L.X	<	F	F	T	F	yes	yes
NL.X	!<	T	T	F	T	yes	yes
NGE.X	!>=	T	F	T	F	yes	yes
GE.X	<=	F	T	F	T	yes	yes

compare-and-branch relations

Arithmetic Operations

The operations supported in hardware are floating-point add, subtract, multiply, divide, and square root. Other operations required by the ANSI/IEEE floating-point standard are provided by software libraries.

The operations explicitly specify the precision of the operation, and round the result to the specified precision at the conclusion of each operation.

A single instruction provides a floating-point multiply with the result fed into a floating-point add. The result is computed as if the multiply is performed to infinite precision, added as if in infinite precision, then rounded. This operation is a particularly good match to the needs of vector linear algebra routines.

Rounding

Rounding is specified within the instructions explicitly, to avoid maintaining explicit state for a rounding mode.

Exceptions

All the mandated floating-point exception conditions cause a trap when they occur; maintenance of sticky and other status bits may be performed using software routines. Because the floating-point inexact exception may be very frequent, this exception only occurs when specified in the instruction explicitly. Arithmetic operations may also specify that all exceptions are to be handled by default, generating special results instead of traps.

Digital Signal Processing

The Terpsichore processor provides a set of operations that maintain the fullest possible use of 64- and 128-bit data paths when operating on lower-precision fixed-point or floating-point vector values. These operations are useful for several application areas, including digital signal processing, image processing, and synthetic graphics. The basic goal of these operations is to accelerate the performance of algorithms that exhibit the following characteristics:

Low-precision arithmetic

The operands and intermediate results are fixed-point values represented in no greater than 64 bit precision. For floating-point arithmetic, operands and intermediate results are of 16, 32, or 64 bit precision.

The use of fixed-point arithmetic permits various forms of operation reordering that are not permitted in floating-point arithmetic. Specifically, commutativity and associativity, and distribution identities can be used to reorder operations. Compilers can evaluate operations to determine what intermediate precision is required to get the specified arithmetic result.

Terpsichore supports several levels of precision, as well as operations to convert between these different levels. These precision levels are always powers of two, and are explicitly specified in the operation code.

Sequential access to data

The algorithms are or can be expressed as operations on sequentially ordered items in memory. Scatter-gather memory access or sparse-matrix techniques are not required.

Where an index variable is used with a multiplier, such multipliers must be powers of two. When the index is of the form: nx+k, the value of n must be a power of two, and the values referenced should have k include the majority of values in the range 0..n-1. A negative multiplier may also be used.

Vectorizable operations

The operations performed on these sequentially ordered items are identical and independent. Conditional operations are either rewritten to use boolean variables or masking, or the compiler is permitted to convert the code into such a form.

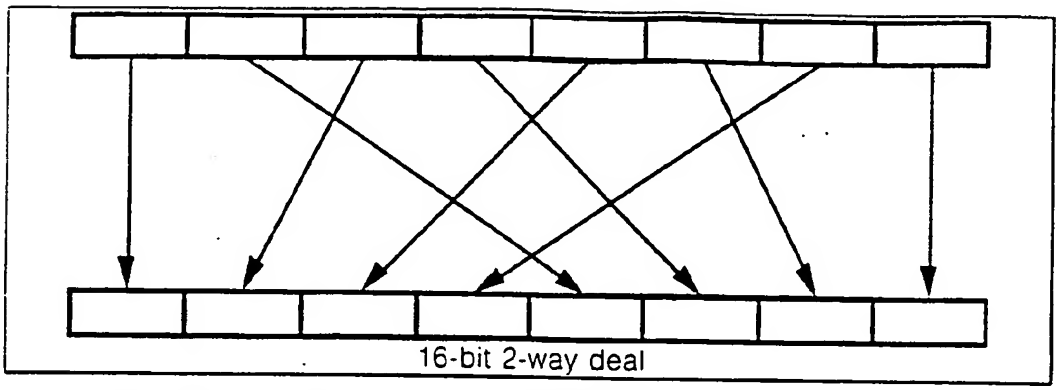
Data-handling Operations

The characteristics of these algorithms include sequential access to data, which permit the use of the normal load and store operations to reference the data. Octlet and hexlet loads and stores reference several sequential items of data, the number depending on the operand precision.

The discussion of these operations is independent of byte ordering, though the ordering of bit fields within octlets and hexlets must be consistent with the ordering used for bytes. Specifically, if big-endian byte ordering is used for the loads and stores, the figures below should assume that index values increase from left to right, and for little-endian byte ordering, the index values increase from right to left. For this reason, the figures indicate different index values with different shades, rather than numbering.

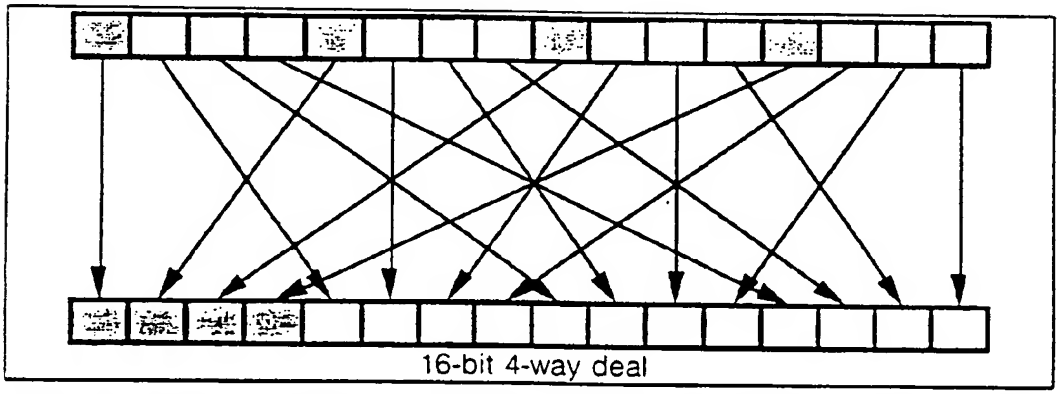
When an index of the nx+k form is used in array operands, where n is a power of 2, data memory sequentially loaded contains elements useful for separate operands. The "deal" instruction divides a hexlet of data up into two octlets, with alternate bit fields of the source hexlet grouped together into the two results. For example, a G.DEAL.16 operation rearranges the source hexlet into two octlets as follows:[1]

[1]An example of the use of a deal can be found in the appendix: Digital Signal Processing Applications: Decimation of Monochrome Image or Decimation of Color Image



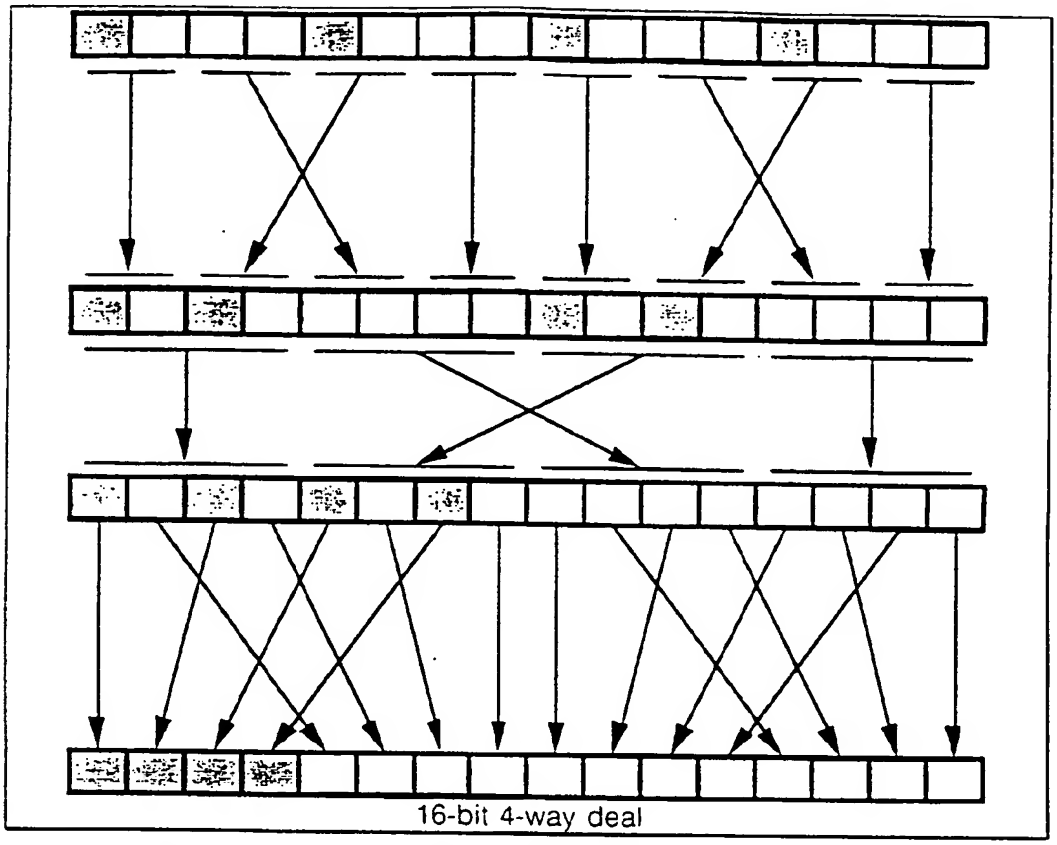
In the deal operation, the source hexlet is specified by two octlet registers, and the two result octlets are specified as a hexlet register pair. (This sounds backwards, and it really is, but it works in practice, because the result is usually used in operations that accept octlet operands. Ideally, the source hexlet should be a register pair, and the result should be two octlet registers.)

The example above directly applies to the case where n is 2. When n is larger, a series of DEAL operations can be used to further subdivide the sequential stream. For example, when n is 4, we need to deal out 4 sets of doublet operands, as shown in the figure below:[2]



This 4-way deal is performed by dealing out 2 sets of quadlet operands, and then dealing each of them out into 2 sets of doublet operands.

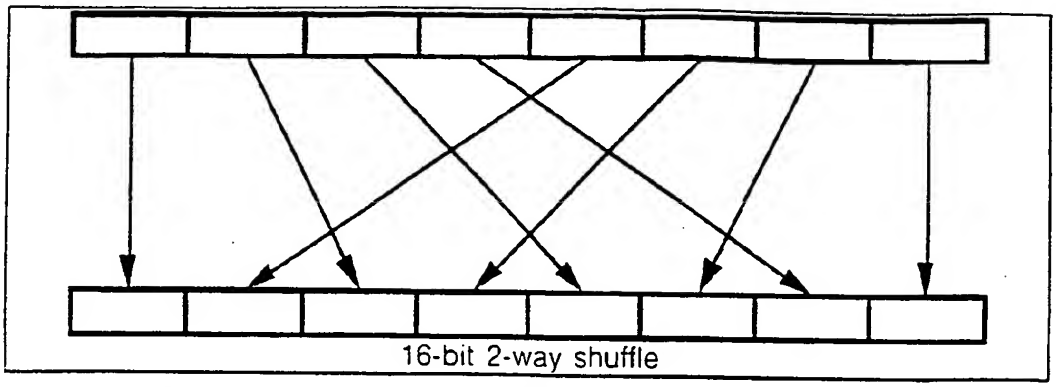
[2]An example of the use of a four-way deal can be found in the appendix: Digital Signal Processing Applications: Conversion of Color to Monochrome



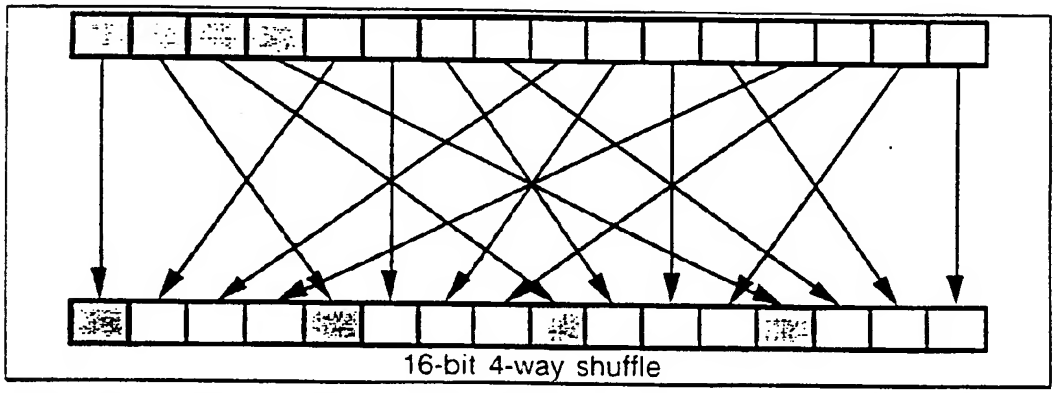
16-bit 4-way deal

There are three rows of arrows shown above. The first row is the result of two G.DEAL.32 operations, each independently dealing 2 sets of pairs of doublets. The result of these two operations is the second row of boxes. The last row is the result of two independent G.DEAL.16 operations, each dealing 2 sets of doublets into register pairs. The middle row of arrows shows the implicit action performed by specifying two non-adjacent registers for the hexlet sources of the G.DEAL.16 operations.

When an array result of computation is accessed with an index of the form nx+k, for n a power of 2, the reverse of the "deal" operation needs to be performed on vectors of results to interleave them for storage in sequential order. The "shuffle" operation interleaves the bit fields of two octlets of results into a single hexlet. For example a G.SHUFFLE.16 operation combines two octlets of doublet fields into a hexlet as follows:

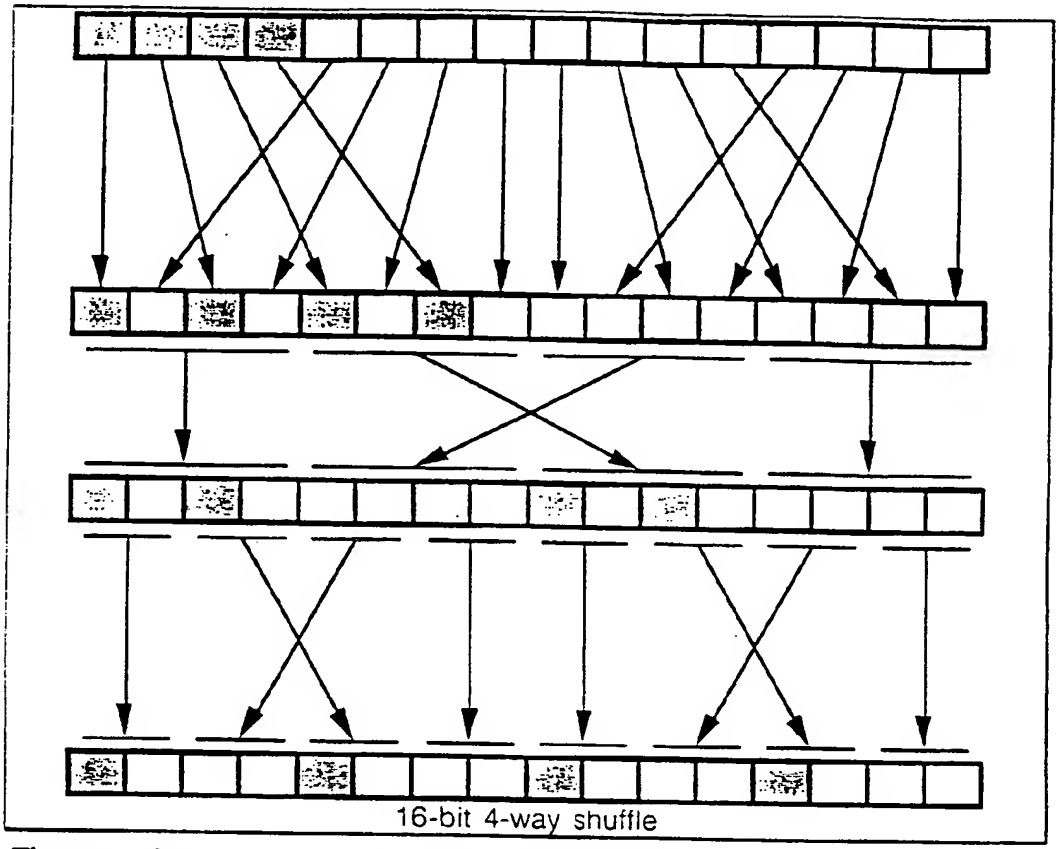


For larger values of n, a series of shuffle operations can be used to combine additional sets of fields, similarly to the mechanism used for the deal operations. For example, when n is 4, we need to shuffle up 4 sets of doublet operands, as shown in the figure below:[3]



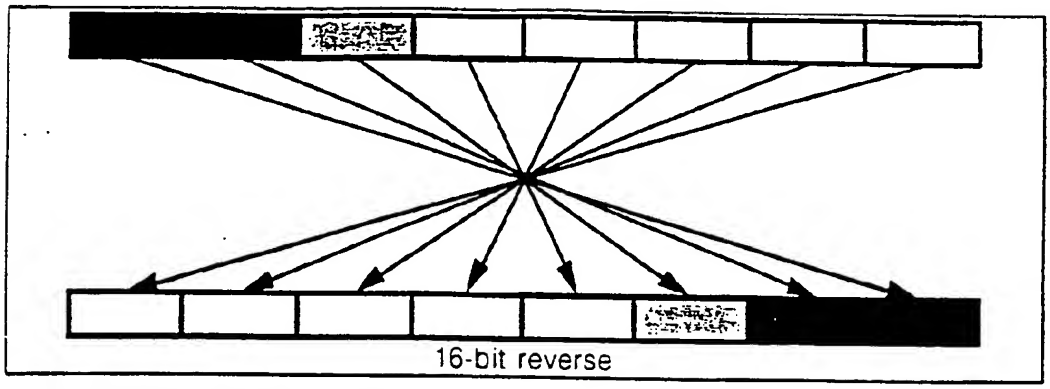
This 4-way shuffle is performed by shuffling up 2 sets of doublet operands, and then shuffling each of them up as 2 sets of quadlet operands.

[3]An example of the use of a four-way shuffle can be found in the appendix: Digital Signal Processing Applications: Conversion of Monochrome to Color



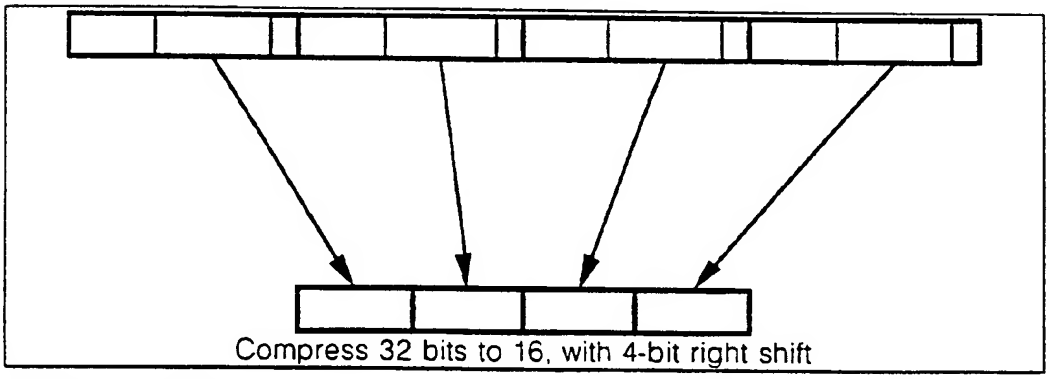
There are three rows of arrows shown above. The first row is the result of two G.SHUFFLE.16 operations, each independently shuffling 2 sets of pairs of doublets. The result of these two operations is the second row of boxes. The last row is the result of two independent G.SHUFFLE.32 operations, each shuffling 2 sets of quadlets into register pairs. The middle row of arrows shows the implicit action performed by specifying two non-adjacent registers for the two octlet sources of the G.SHUFFLE.32 operations.

When the index of a source array operand or a destination array result is negated, or in other words, if of the form nx+k where n is negative, the elements of the array must be arranged in reverse order. The "swap" operation reverses the order of the bit fields in a hexlet. For example, a G.SWAP.16 operation reverses the doublets within a hexlet:



Variations of the deal and shuffle operations are also useful for converting from one precision to another. This may be required if one operand is represented in a different precision than another operand or the result, or if computation must be performed with intermediate precision greater than that of the operands, such as when using an integer multiply.

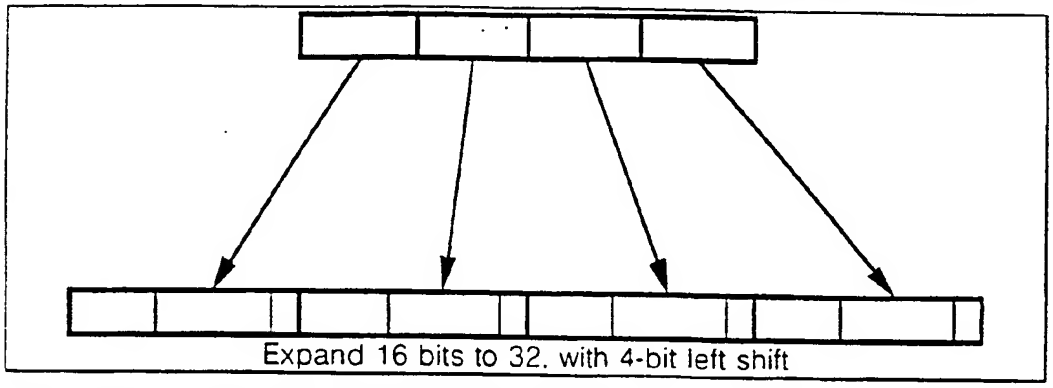
When converting from a higher precision to a lower precision, specifically when halving the precision of a hexlet of bit fields, half of the data must be discarded, and the bit fields packed together. The "compress" operation is a variant of the "deal" operation, in which the operand is a hexlet, and the result is an octlet. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, a selection of bits 19..4 of each quadlet in a hexlet is performed by the G.COMPRESS.16,4 operation:



When converting from lower-precision to higher-precision, specifically when doubling the precision of an octlet of bit fields, one of several techniques can be used, either multiply, expand, or shuffle. Each has certain useful properties. In the discussion below, m is the precision of the source operand.

The multiply operation, described in detail below, automatically doubles the precision of the result, so multiplication by a constant vector will simultaneously double the precision of the operand and multiply by a constant that can be represented in m bits.

An operand can be doubled in precision and shifted left with the "expand" operation, which is essentially the reverse of the "compress" operation. For example the G.EXPAND.16,4 expands from 16 bits to 32, and shifts 4 bits left:



The "shuffle" operation can double the precision of an operand and multiply it by 1 (unsigned only), 2^m or 2^m+1, by specifying the sources of the shuffle operation to be a zeroed register and the source operand, the source operand and zero, or both to be the source operand. When multiplying by 2m, a constant can be freely added to the source operand by specifying the constant as the right operand to the shuffle.

Arithmetic Operations

The characteristics of the algorithms that affect the arithmetic operations most directly are low-precision arithmetic, and vectorizable operations. The fixed-point arithmetic operations provided are most of the functions provided in the standard integer unit, except for those that check conditions. These functions include add, subtract, bitwise boolean operations, shift, set on condition, and multiply, in forms that take packed sets of bit fields of a specified size as operands. The floating-point arithmetic operations provided are as complete as the scalar floating-point arithmetic set. The result is generally a packed set of bit fields of the same size as the operands, except that the fixed-point multiply function intrinsically doubles the precision of the bit field.

Conditional operations are provided only in the sense that the set on condition operations can be used to construct bit masks that can select between alternate vector expressions, using the bitwise boolean operations. All instructions operate over the entire octlet or hexlet operands, and produce a hexlet result. The sizes of the bit fields supported are always powers of two.

Galois Field Operations

Terpsichore provides a general software solution to the most common operations required for Galois Field arithmetic. The instruction provided is a polynomial divide, with the polynomial specified as one register operand. The result of a specified number of division steps, expressed as a register pair, is the result of the instruction. This instruction can be used to perform CRC generation and

checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding.

Software Conventions

The following section describes software conventions which are to be employed at sofware module boundaries, in order to permit the combination of separately compiled code and to provide standard interfaces between application, library and system software. Register usage and procedure call conventions may be modified, simplified or optimized when a single compilation encloses procedures within a compilation unit so that the procedures have no external interfaces. For example, internal procedures may permit a greater number of register-passed parameters, or have registers allocated to avoid the need to save registers at procedure boundaries, or may use a single stack or data pointer allocation to suffice for more than one level of procedure call.

Register Usage

All Terpsichore registers are identical and general-purpose; there is no dedicated zero-valued register, and no dedicated floating-point registers. However, some procedure-call-oriented instructions imply usage of registers zero (0) and one (1) in a manner consistent with the conventions described below. By software convention, the non-specific general registers are used in more specific ways.

register number	usage	how saved
0	link	caller
1	dp	caller
2-9	parameters	caller
10-31	temporary	caller
32-61	saved	callee
62	fp, when required	callee
63	sp	callee

register usage

At a procedure call boundary, registers are saved either by the caller or callee procedure, which provides a mechanism for leaf procedures to avoid needing to save registers. Optimizers may choose to allocate variables into caller or callee saved registers depending on how their lifetimes overlap with procedure calls.

The dp register points to a small (4 kilobyte) array of pointers, literals, and statically-allocated variables, which is used locally to the procedure. The uses of the dp register are similar to the Mips use of the gp register, except that each procedure may have a different value, which expands the space addressable by small offsets from this pointer. This is an important distinction, as the offset field of Terpsichore load and store instructions are only 12 bits. The compiler may use additional registers and/or indirect pointers to address larger regions.

This mechanism also permits code to be shared, with each static instance of the dp region assigned to a different address in memory. In conjunction with position-independent or pc-relative branches, this allows library code to be dynamically relocated and shared between processes.

Procedure Calling Conventions

Procedure parameters are normally allocated in registers, starting from register 2 up to register 9. These registers hold up to 8 parameters, which may each be of any size from one byte to eight bytes, including single-precision and double-precision floating-point parameters. Quad-precision floating-point parameters require an aligned pair of registers. The C varargs.h or stdarg.h conventions may require saving registers into memory (this is not necessarily so, but some semi-portable semi-conventions such as _doprnt would break otherwise). Procedure return values are also allocated in registers, starting from register 2 up to register 9.

There are several data structures maintained in registers for the procedure calling conventions: link, sp, dp, fp. The link register contains the address to which the callee should return to at the conclusion of the procedure.

The sp register is used to form addresses to save parameter and other registers, maintain local variables, i.e., data that is allocated as a LIFO stack. For procedures that require a stack, normally a single allocation is performed, which allocates space for input parameters, local variables, saved registers, and output parameters all at once. The sp register is always 16-byte aligned.

The dp register is used to address pointers, literals and static variables for the procedure. The newpc register is loaded with the entry point of the procedure, and the newdp register is loaded with the value of the dp register required for the procedure. This mechanism provides for dynamic linking, by initially filling in the link and dp fields in the data structure to point to the dynamic linker. The linker can use the current contents of the link and/or dp registers to determine the identity of the caller and callee, to find the location to fill in the pointers and resume execution.

The fp register is used to address the stack frame when the stack size varies during execution of a procedure, such as when using the GNU alloca function. When the stack size can be determined at compile time, the sp register is used to address the stack frame and the fp register may be used for other general purposes as a callee-saved register.

Typical static-linked, intra-module calling sequence:

caller or callee (non-leaf):

```
A.ADDI          sp.-size
S.64            link.off(sp)
... (using same dp as caller)
B.LINK.I        callee
...
L.64            link.off(sp)
```

```
        A.ADDI          sp.size
        B               link
```

callee (leaf):

```
        ... (using dp)
        B               link
```

Typical dynamic-linked, inter-module calling sequence:

caller or callee (non-leaf):

```
        A.ADDI          sp.-size
        S.128           linkdp.off(sp)
        ... (using dp)
        L.128           linkdp.off(dp)
        B.LINK          link.link
        L.128           linkdp.off(sp)
        ... (using dp)
        A.ADDI          sp.size
        B               link
```

callee (leaf):

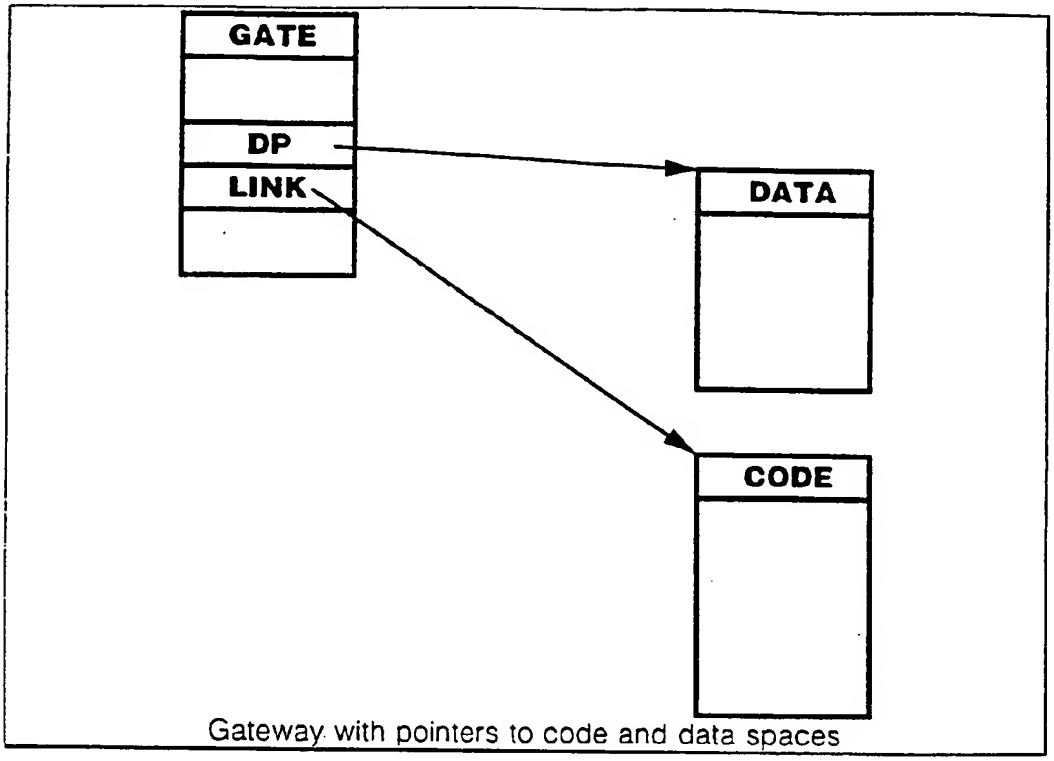
```
        ... (using dp)
        B               link
```

The load instruction is required in the caller following the procedure call to restore the dp register. A second load instruction also restores the link register, which may be located at any point between the last procedure call and the branch instruction which returns from the procedure.

System and Privileged Library Calls

It is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, we prefer to use a modified procedure call in which the process privilege level is quietly raised to the required level. In to provide this mechanism safely, interaction with the virtual memory system is required.

Such a routine must not be entered from anywhere other than its legitimate entry point, otherwise the sudden access to a higher privilege level might be taken advantage of. The branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that the data pointer is properly set. To ensure this, the branch-through-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway contains the virtual address of the entry point of the procedure and the virtual address to be loaded into the data pointer. A gateway can only exist in regions of the virtual address space designated to contain them, to ensure that a gateway cannot be forged.



Gateway with pointers to code and data spaces

Similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities, so a procedure may freely branch to a less-privileged code address. However, in certain, though perhaps rare, cases, it would be useful to have highly privileged code call less-privileged routines. As an example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. In such a case, a return from a procedure actually requires an increase in privilege, which must be carefully controlled. Again, a branch-through-gateway instruction can be used, this time in the instruction following the call, to raise the privilege again in a controlled fashion. In such a case, special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame that may be manipulated by the less-privileged routine.

Typical dynamic-linked, inter-gateway calling sequence:

caller:

```
...
S.128          linkdp,off(sp)
...
B.GATE         linkdp,off(dp)
L.128          linkdp,off(sp)
```

callee (non-leaf):

```
S.64            sp.off(dp)
L.64            sp.off(dp)
S.128           link.dp.off(sp)
... (using dp)
L.128           link.dp.off(sp)
L.64            sp.off(dp)
B.DOWN          link
```

callee (leaf):

```
... (using dp)
B.DOWN          link
```

The callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, except as a region to receive parameters held in memory.

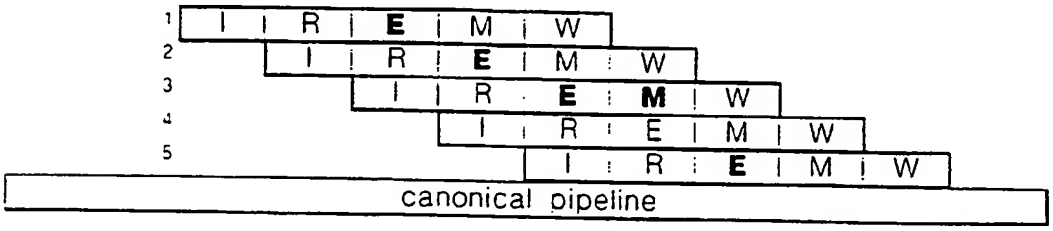
Pipeline Organization

Euterpe performs all instructions as if executed one-by-one, in-order, with precise exceptions always available. Consequently, code which ignores the subsequent discussion of Euterpe pipeline implementations will still perform correctly. However, the highest performance of the Euterpe processor is achieved only by matching the ordering of instructions to the characteristics of the pipeline. In the following discussion, the general characteristics of all Euterpe implementations preceeds discussion of specific choices for specific implementations.

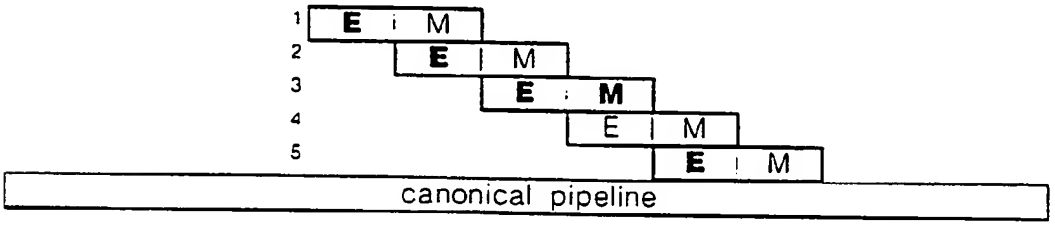
Classical Pipeline Structures

Pipelining in general refers to hardware structures that overlap various stages of execution of a series of instructions so that the time required to perform the series of instructions is less than the sum of the times required to perform each of the instructions separately. Additionally, pipelines carry to connotation of a collection of hardware structures which have a simple ordering and where each structure performs a specialized function.

The diagram below shows the timing of what has become a canonical pipeline structure for a simple RISC processor, with time on the horizontal axis increasing to the right, and successive instructions on the vertical axis going downward. The stages I, R, E, M, and W refer to units which perform instruction fetch, register file fetch, execution, data memory fetch, and register file write. The stages are aligned so that the result of the execution of an instruction may be used as the source of the execution of an immediately following instruction, as seen by the fact that the end of an E stage (bold in line 1) lines up with the beginning of the E stage (bold in line 2) immediately below. Also, it can be seen that the result of a load operation executing in stages E and M (bold in line 3) is not available in the immediately following instruction (line 4), but may be used two cycles later (line 5); this is the cause of the load delay slot seen on some RISC processors.



In the diagrams below, we simplify the diagrams somewhat by eliminating the pipe stages for instruction fetch, register file fetch, and register file write, which can be understood to preceed and follow the portions of the pipelines diagrammed. The diagram above is shown again in this new format, showing that the canonical pipeline has very little overlap of the actual execution of instructions.



A superscalar pipeline is one capable of simulatenously issuing two or more instructions which are independent, in that they can be executed in either order and separately, producing the same result as if they were executed serially. The diagarm below shows a two-way superscalara processor, where one instruction may be a register-to-register operation (using atage E) and the other may be a register-to-register operation (using atage A) or a memory load or store (using stages A and M).